Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Uma Arquitectura para a Monitorização de Computações Paralelas e Distribuídas

Vítor Manuel Alves Duarte

Dissertação apresentada para obtenção do Grau de Doutor
em Informática, pela Universidade Nova de Lisboa,
Faculdade de Ciências e Tecnologia.

Lisboa
2003

Esta tese foi orientada pelo Doutor José Alberto Cardoso e Cunha, Professor no Departamento de Informática, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Agradecimentos

A elaboração deste trabalho não seria possível sem a colaboração e compreensão de várias pessoas.

Ao meu orientador, José Cardoso e Cunha, pela orientação neste trabalho, apoio e motivação, sem os quais esta dissertação não teria sido concluída.

Aos alunos que co-orientei, e que contribuíram com as suas implementações para partes deste trabalho.

A todos os meus colegas do Departamento de Informática da FCT/UNL, pelo ambiente criado e amizade, em particular para a Cecília Gomes, o Pedro Medeiros, o Rui Marques e o João Lourenço.

A todos os colegas, participantes nos projectos relacionados com este trabalho, em particular ao Professor Jacques-Chassin de Kergommeaux.

À minha família, pelo apoio dado e grande compreensão, ao longo destes anos.

Quero aqui deixar expressos os meus agradecimentos a todos aqueles que, directa ou indirectamente, contribuíram para a realização deste trabalho.

Resumo

O recurso à monitorização do comportamento dos programas durante a execução é necessário em diversos contextos de aplicação. Por exemplo, para verificar a utilização dos recursos computacionais durante a execução, para calcular métricas que permitam melhor definir o perfil da aplicação ou para melhor identificar em que pontos da execução estão as causas de desvios do comportamento desejado de um programa e, noutros casos, para controlar a configuração da aplicação ou do sistema que suporta a sua execução.

Esta técnica tem sido aplicada, quer no caso de programas sequenciais, quer se trate de programas distribuídos. Em particular, no caso de computações paralelas, dada a complexidade devida ao seu não determinismo, estas técnicas têm sido a melhor fonte de informação para compreender a execução da aplicação, quer em termos da sua correcção, quer na avaliação do seu desempenho e utilização dos recursos computacionais.

As principais dificuldades no desenvolvimento e na adopção de ferramentas de monitorização, prendem-se com a complexidade dos sistemas de computação paralela e distribuída e com a necessidade de desenvolver soluções específicas para cada plataforma, para cada arquitectura e para cada objectivo. No entanto existem funcionalidades genéricas que, se presentes em todos os casos, podem ajudar ao desenvolvimento de novas ferramentas e à sua adaptação a diferentes ambientes computacionais.

Esta dissertação propõe um modelo para suportar a observação e o controlo de aplicações paralelas e distribuídas (*DAMS - Distributed Applications Monitoring System*). O modelo define uma arquitectura abstracta de monitorização baseada num núcleo mínimo sobre o qual assentam conjuntos de serviços que realizam as funcionalidades pretendidas em cada cenário de utilização. A sua organização em camadas de abstracção e a capacidade de extensão modular, permitem suportar o desenvolvimento de conjuntos de funcionalidades que podem ser partilhadas por distintas ferramentas. Por outro lado, o modelo proposto facilita o desen-

volvimento de ferramentas de observação e controlo, sobre diferentes plataformas de suporte à execução.

Nesta dissertação, são apresentados exemplos da utilização do modelo e da infraestrutura que o suporta, em diversos cenários de observação e controlo. Descreve-se também a experimentação realizada, com base em protótipos desenvolvidos sobre duas plataformas computacionais distintas.

Abstract

In many distinct contexts, the need for monitoring of program execution is recognized as an important activity. For example, to support resource management and administration, to obtain information on the performance behavior of the application, to help in supporting program testing and debugging, and, in other cases, to support adaptative control of ongoing computations.

Monitoring techniques have been applied, both concerning sequential and distributed programs. Namely, regarding parallel computations, due to their intrinsic non-determinism, monitoring techniques are fundamental to provide relevant information to help understanding program behavior; in terms of its correctness, its performance and their effectiveness of resource utilization.

The main difficulties posed by the practical development and usage of monitoring tools, are related to the complexity of parallel and distributed computing systems. Most existing tools are too much dependent upon specific application scenarios and their runtime support platforms. However, one can try to identify a set of generic functionalities which could ease the development of new tools, as well as their application in distinct computing environments.

In this dissertation, a model is proposed for an abstract monitoring architecture and its supporting infrastructure (DAMS – Distributed Applications Monitoring System). The DAMS approach relies upon a minimal core of generic functionalities plus an open and extensible set of services. Each service provides the required observation and control functionalities to support each specific application scenario.

The DAMS abstractions and module organization contribute to ease the development of a diversity of monitoring functionalities that may be shared by distinct tools. The implementation of its architecture provides a reasonable portability level to the DAMS infrastructure

on distinct computing platforms.

In this dissertation, examples of applications of the DAMS model are presented, for distinct scenarios involving observation and control of parallel and distributed computations. A discussion is also included concerning the experimental development of DAMS prototypes.

Sommaire

L'utilisation des techniques de collecte de traces et de controle d' exécution de programmes est nécessaire dans plusieurs contextes d'application. Par exemple, si on veut vérifier l' utilisation des resources pendant l'exécution, si on veut obtenir des mesures pour aider à l'analyse de certaines caractéristiques de chaque application ou pour aider à l'identification d'un comportement particulier de l'application, ou encore, pour le controle de la configuration de l'application, elle-même, pendant l'exécution.

Les techniques de surveillance sont nécessaires pour l'exécution séquentielle, bien que pour l'exécution parallèle. Le développement d'applicatoins parallèles correctes et efficaces etant très complex, il faut utiliser des techniques de surveillance, pour obtenir des informations qui peuvent aider à l'identification des problèmes de correction et de performance.

Les obstacles pour le développment et utilisation des outils de surveillance sont liés au complexité des programmes parallèles et distribuées. Il faut aussi être capable de développer des solutions spécifiques et biens adaptées à chaque plateforme et à chaque but. Il est possible, quand même, d'identifier des fonctions, on peut dire presques géneriques, qui peuvent aider au développement des nouveaux outils, et biens adaptées à différentes plateformes.

Cette thèse propose um modèle pour supporter l'observation et le controle des applications parallèles et distribuées (*DAMS - Distributed Applications Monitoring System*). Le modèle propose une architecture abstraite pour la surveillance, qui se développe ao tour d'un noyau minimaliste. Au dessous de ce nucleous on peut définir des différents services pour supporter les besoins de chaque cénaire d'application. L'organisation modulaire, basée en plusieurs couches d'abstraction, et les possibilitées d'extension incrementale, aide à le développement de plusieurs fonctionalités, qui on peut faire partager parmi des différentes outils. Aussi, le modèle proposé aide à le développement de nouveaux outils, pour des différentes plateformes.

Dans cette thèse, on présente des examples de l’utilisation du modèle et de l’architecture qui le supporte, dans plusieurs cénaires d’ application. On décrit aussi les travaux experimentales qui ont étés conduits, pour valider les différents prototypes développés.

Conteúdo

1 Introdução **1**

1.1 Motivação 3

1.2 Enquadramento do trabalho 6

1.2.1 Desenvolvimento de *software* para aplicações paralelas 7

1.2.2 Monitorização para suportar a depuração distribuída 8

1.2.3 Monitorização de distintos níveis de abstracção 9

1.2.4 Monitorização para suportar observação *online* e controlo dinâmico 10

1.3 Contribuições 10

1.4 Organização da dissertação 11

2 Fundamentos da Monitorização de Computações Distribuídas **13**

2.1 Introdução 13

2.2 Computações paralelas e distribuídas 14

2.3 Observação de estados globais de computações distribuídas 25

2.4 Monitorização de computações paralelas e distribuídas 27

2.4.1 Abordagens de monitorização 27

2.4.2 Funcionalidades para a monitorização de programas distribuídos . . 29

2.5 Modelo teórico de um sistema de monitorização 37

2.5.1 Introdução 37
2.5.2 Sistema de monitorização 38
2.5.3 Perturbação devida à observação 40
2.5.4 A dimensão temporal e a completude da observação 43
2.5.5 A dimensão temporal e a adequação da observação 45
2.6 Conclusões 46

3 Arquitecturas de Suporte à Monitorização 49

3.1 Arquitectura do sistema de monitorização 49
3.2 Requisitos para a concepção de arquitecturas de monitorização 50
3.2.1 Introdução 50
3.2.2 Abordagens de realização da arquitectura 52
3.2.3 Conclusão 54
3.3 Sistemas monolíticos de monitorização 54
3.3.1 Interacção baseada em ficheiros de traço 54
3.3.2 Interfaces para a instrumentação e controlo 56
3.3.3 Interfaces e arquitecturas de suporte à monitorização 57
3.4 A motivação para o desenvolvimento do modelo DAMS 60
3.4.1 Introdução 60
3.4.2 O desenvolvimento incremental do modelo *DAMS* 60
3.5 Conclusões 61

4 O Modelo DAMS 63

4.1 Introdução 63
4.2 Primeira abordagem 65

4.3 Objectivos e requisitos 68

4.4 Dimensões e características do modelo 71

4.4.1 Arquitectura abstracta 74

4.5 Serviços 75

4.5.1 Interacção com os sensores/actuadores 77

4.5.2 Interacção com os clientes 79

4.5.3 Interacção com o núcleo e outros serviços 80

4.6 O núcleo da arquitectura DAMS 80

4.6.1 Gestão de serviços 82

4.6.2 Gestão da máquina virtual de monitorização 84

4.6.3 Canais de eventos 85

4.6.4 Interface com a plataforma de suporte 88

4.7 Exemplos de serviços 91

4.7.1 Serviço de traços 92

4.7.2 Serviço de controlo 97

4.8 Conclusão 100

5 Implementação da arquitectura DAMS 101

5.1 Introdução 101

5.2 Arquitectura da DAMS 102

5.2.1 Entidades 103

5.2.2 Interface com a plataforma 105

5.3 O primeiro protótipo sobre o PVM 112

5.4 O segundo protótipo da DAMS 114

5.4.1 Interface com o ORBit 116

5.5 Realização do núcleo DAMS 117
5.5.1 Implementação das operações do núcleo DAMS 118
5.5.2 Gestão de máquinas 119
5.5.3 Gestão de serviços 121
5.5.4 Canais de eventos 123
5.6 Conclusão 126

6 Cenários de Utilização 129

6.1 Cenários estudados 129
6.2 Consola de gestão 131
6.3 Observação 132
6.3.1 Serviço de traço 132
6.3.2 Monitorização de aplicações PVM-Prolog 137
6.3.3 Monitorização de aplicações MPI 145
6.3.4 Monitorização de aplicações Java 150
6.4 Controlo para depuração 154
6.4.1 Serviço para depuração 155
6.4.2 Serviço central para depuração 157
6.5 Controlo dinâmico (*steering*) 159
6.6 Conclusão 166

7 Conclusões 169

7.1 Avaliação 169
7.2 Direcções de trabalho futuro 170

Bibliografia 173

Lista de Figuras

2.1 Organização de um sistema de monitorização 39

3.1 Hierarquia de camadas de suporte ao monitor. 50

3.2 *a)* Sistema monolítico; *b)* Interface através de traço 55

3.3 Interfaces para controlo e instrumentação 56

3.4 Separando a monitorização das ferramentas 58

3.5 Infraestrutura de monitorização flexível 58

4.1 Arquitectura do primeiro modelo DAMS 66

4.2 Relação da DAMS com a aplicação 72

4.3 Modelo DAMS 72

4.4 Modelo de arquitectura DAMS 74

4.5 Esquema de um serviço 77

4.6 Um serviço como intermediário dos sensores ou actuadores 78

4.7 Sensor ou actuador integrando-se como um serviço 78

4.8 Sensor ou actuador cliente de um serviço 78

4.9 Esquema do núcleo 81

4.10 Núcleo da DAMS 82

4.11 Rede de canais de eventos 87

4.12 Um canal para traços 93

4.13 Monitorização usando uma rede de serviços de traço 94

4.14 Um serviço para depuração na DAMS 98

5.1 Arquitectura da DAMS 102

5.2 Realização do núcleo DAMS 106

5.3 Implementação das interacções em serviços da DAMS 106

5.4 Níveis entre clientes e servidores de serviços 116

6.1 Cenários estudados 130

6.2 Implementação do serviço de traço 135

6.3 Monitorização do PVM-Prolog 138

6.4 Instrumentação no PVM-Prolog 141

6.5 Instrumentação de PVM-Prolog na nova DAMS 143

6.6 Monitorização de uma máquina JVM 150

6.7 Jumpshot mostrando o traço obtido de uma execução Java. 152

6.8 Monitorização de várias JVM 154

6.9 Um serviço para depuração usando o `gdb` 156

6.10 Um serviço para depuração global na DAMS 159

6.11 Arquitectura do sistema de *steering* dedicado 160

6.12 Aspecto do visualizador de *steering* ligado a uma ilha. 162

6.13 *Steering* de AG utilizando o primeiro modelo da DAMS 163

6.14 *Steering* de AG utilizando o novo modelo da DAMS 165

6.15 *Steering* de AG utilizando a DAMS (com visualizador autónomo) 165

Capítulo 1

Introdução

Nos últimos quarenta anos, tem havido uma constante procura de soluções para melhorar o desempenho das aplicações, tirando partido de arquitecturas de computadores com múltiplas unidades de processamento. Ao longo deste período, uma grande diversidade de arquitecturas foi desenvolvida, explorando a execução paralela, quer dentro das próprias unidades de processamento, quer em sistemas multiprocessadores de memória fisicamente partilhada ou de memória distribuída, até às redes locais e aos agregados de computadores[1]. Estes desenvolvimentos nas arquitecturas que suportam a computação paralela têm tido enorme influência, em múltiplos níveis da evolução tecnológica dos sistemas informáticos, incluindo o *hardware* dos microprocessadores actuais e das estruturas de interligação de computadores, as abstracções de suporte da concorrência, paralelismo e distribuição a nível dos sistemas de operação, os paradigmas e modelos de programação e os ambientes e ferramentas para o desenvolvimento de *software*. Esta influência é justificável, na medida em que, desde muito cedo, a comunidade científica e também a comunidade industrial se aperceberam do interesse da exploração de plataformas de computação paralela, de modo a possibilitar aplicações de grande interesse estratégico, quer no domínio teórico, quer com impacto industrial, envolvendo, em geral, a simulação de processos complexos nas Ciências e Engenharias e outras aplicações nas mais diversas áreas de actividade humana.

Com a evolução das redes de computadores, alargaram-se as oportunidades para se explorarem, como plataformas computacionais para a execução de aplicações, tais arquitecturas paralelas e distribuídas, constituídas por uma diversidade de unidades processadoras,

[1] *Clusters.*

mas aparecendo aos utilizadores como uma máquina virtual unificada, oferecendo adequados graus de transparência.

Apesar de todos estes desenvolvimentos, o impacto dos sistemas de computação paralela tem sido travado sobretudo pelas dificuldades que se colocam no desenvolvimento de *software* que seja eficaz na exploração do paralelismo, de modo a compensar o esforço de desenvolvimento exigido aos programadores. O desenvolvimento de aplicações paralelas para tais arquitecturas revela-se muito mais complexo que do que o habitual em aplicações sequenciais.

Estas dificuldades começam logo pela necessidade de, dado um problema, se analisarem as possibilidades da sua decomposição em unidades bem definidas, que sejam susceptíveis de execução paralela. Por um lado, nem todos os problemas permitem uma decomposição fácil em unidades independentes entre si, colocando a necessidade de planear a cooperação entre os componentes da aplicação, baseada numa diversidade de modelos de comunicação e de sincronização, os quais dificultam esta tarefa. Por outro lado, mesmo que atingida uma 'razoável' solução para a decomposição da aplicação, do ponto de vista lógico, há que estabelecer a correspondência entre as unidades lógicas da aplicação e os processadores reais que suportarão a sua execução. As características específicas de cada arquitectura *hardware*, bem como do *software* do sistema de operação, do compilador e do próprio modelo de programação utilizado, podem influenciar decisivamente o desempenho que se pode obter, ao executar uma dada aplicação sobre uma plataforma computacional paralela. Infelizmente, dada a complexidade dos factores envolvidos, que influenciam o desempenho final obtido, a maioria dos sistemas depende criticamente do apoio dado por ferramentas e ambientes de desenvolvimento que permitem ensaiar e avaliar experimentalmente o comportamento das aplicações. Procura-se, assim, ajudar o programador a considerar possíveis alternativas de decomposição das aplicações, para as ajustar às características da plataforma computacional (*hardware* e *software*).

No entanto, àquelas dificuldades, acrescem outras, relacionadas com a necessidade de o programador garantir a correcção dos programas. Dados a concorrência e o paralelismo, surge uma explosão combinatória do número de execuções possíveis, que deveriam, teoricamente, ser analisadas, para se verificar a correcção do seu comportamento. Acresce que, na presença de arquitecturas de sistemas distribuídos, como os agregados de processadores ou a redes globais de computadores, há que lidar com as dificuldades de compreensão do compor-

tamento dos sistemas distribuídos assíncronos (ausência de estado global, desconhecimento dos tempos limite para a transmissão de mensagens, ocorrências de falhas). Nestes contextos, a tarefa de compreender o comportamento de um programa, identificando e localizando os possíveis erros, torna-se muito difícil.

Nos últimos vinte anos, desde que começaram a surgir as primeiras arquitecturas de multiprocessadores comerciais, tem-se verificado intensa investigação em torno de ferramentas e ambientes para a avaliação do comportamento de aplicações paralelas, nas suas múltiplas dimensões, da correcção e do desempenho. Muitas ferramentas têm sido desenvolvidas, cada vez mais poderosas, para os mais variados sistemas e modelos de programação. Nalguns casos particulares, apresentam-se ambientes de desenvolvimento integrando uma multiplicidade de ferramentas, que se complementam entre si, no suporte às mais variadas tarefas de desenvolvimento, tais como, interacção com o utilizador, análise estática e dinâmica do programa, teste e depuração, simulação, activação e monitorização da execução, com equilíbrio dinâmico da carga[2], visualização). O êxito destes ambientes integrados tem sido, na maior parte dos casos, comprometido pelas dificuldades de interacção entre ferramentas desenvolvidas separadamente e que frequentemente se baseiam em distintos modelos computacionais e de suporte à execução.

Face a este panorama, tem continuado em aberto a questão de definir um conjunto de conceitos e mecanismos que, de uma forma generalizada, possam dar suporte àquelas ferramentas, nos mais variados ambientes de desenvolvimento, plataformas computacionais e arquitecturas paralelas e distribuídas.

1.1 Motivação

Neste trabalho, centrou-se a atenção nas funcionalidades para suportar a observação e o controlo de aplicações paralelas e distribuídas. O papel desempenhado pelos sistemas de observação e controlo, aqui referidos genericamente como *monitores* ou *sistemas de monitorização*, é o de possibilitar a recolha de informação relevante sobre o comportamento das computações distribuídas a diversos níveis, desde o *hardware* até à própria aplicação, de modo a disponibilizar essa informação ao utilizador ou a ferramentas específicas, como as

[2] *load balancing.*

acima mencionadas.

Um sistema de monitorização deve suportar a recolha de informação e poder também efectuar alterações nos vários componentes do sistema monitorizado. As seguintes fases são habitualmente suportadas:

1. a recolha de informação relevante, nos múltiplos nós da arquitectura distribuída, nos quais se executem processos componentes da aplicação;

2. o processamento e armazenamento intermédios da informação coligida e a sua possível transferência para os nós contendo outras ferramentas para posterior análise, armazenamento e apresentação ao utilizador, permitindo ainda a possibilidade de provocar alterações na aplicação ou na própria plataforma computacional subjacente.

Uma diversidade de abordagens têm sido propostas, dependendo dos objectivos do utilizador e das facilidades disponíveis no sistema distribuído, deixando em aberto múltiplas opções, quando se trata de conceber um novo sistema de monitorização ou seleccionar um sistema existente. Por exemplo:

- Pretende-se a observação da aplicação durante a sua execução ou apenas a análise do seu comportamento após a execução (monitorização *on-line* vs. *post-mortem*)?

- Qual o grau de perturbação da aplicação, originado pelo processo de observação, que se considera tolerável, em cada caso?

- A que níveis de abstracção se pretende oferecer a possibilidade de observação e de controlo?

- Quais as funcionalidades que a plataforma computacional e a arquitectura do sistema distribuído disponibilizam para esse efeito? Em particular, dispõe-se de recursos computacionais (processador, memória, canais de comunicação) dedicáveis às tarefas de monitorização ou temos de os partilhar com a própria aplicação?

Observação *post-mortem*. Para dar resposta a estas questões, têm surgido muitos sistemas de monitorização, os quais, no entanto, em geral, estão rigidamente associados a plataformas e a ferramentas específicas. Para além dessas dependências, uma grande parte das soluções

de monitorização de computações paralelas baseiam-se em ferramentas que produzem formatos específicos para os registos de eventos que descrevem os traços da execução, visando representações compactas ou o processamento eficiente dos registos dos eventos. Em consequência, a forma habitual de interagir com tais ferramentas é através de procedimentos de conversão dos formatos dos traços. Mesmo que se definisse um formato normalizado "universal" para os traços, o tipo de interacções entre ferramentas ficaria limitado, em geral, a métodos de observação e processamento *post-mortem*.

No ciclo tradicional de desenvolvimento de programas paralelos, a recolha de dados suporta a obtenção de indicadores do comportamento das computações, os quais são habitualmente passados a ferramentas que, *a posteriori*, os analisam e permitem a um utilizador tirar conclusões sobre a adequação do comportamento ou, pelo contrário, a necessidade de o corrigir, através de modificações do programa, a avaliar em sucessivas iterações daquele ciclo. Esta metodologia é adequada para o desenvolvimento de aplicações relativamente "estáticas" e executando-se sobre plataformas computacionais que não exibem variações apreciáveis na sua configuração ou no seu comportamento durante a execução.

Observação *online* e controlo dinâmico. Em contraste, no caso de aplicações sujeitas a modificações dinâmicas do seu comportamento e executadas sobre sistemas distribuídos heterogéneos, cujos recursos computacionais estejam também sujeitos a variações dinâmicas, por exemplo, na sua acessibilidade ou no seu desempenho, os métodos de observação *post-mortem* tornam-se menos adequados. Em alternativa, o ciclo de desenvolvimento passa a dispor de uma fase de recolha de dados, seguida da sua análise e observação *online*, conduzindo, possivelmente a alterações no programa, efectuadas dinamicamente, isto é, durante a sua execução. Este tipo de funcionamento, permitindo formas de controlo dinâmico, adaptável em função das variações de comportamento dinâmico da aplicações, bem como das plataformas computacionais subjacentes, é cada vez mais necessário nos nossos dias. No entanto, não é de excluir a possibilidade de um ciclo de desenvolvimento combinando fases de observação *post-mortem* ou *offline* com fases de observação e controlo dinâmicos.

Numa perspectiva diferente da anterior, a tendência para abranger, num mesmo ambiente integrado, as fases de desenvolvimento de uma aplicação e de observação e controlo da sua execução, surgiu associada a muitos sistemas de simulação, em aplicações industriais (por exemplo, na indústria automóvel ou aeroespacial), permitindo a um engenheiro monitorizar

a execução de uma simulação complexa, visualizando o seu comportamento *online* e ajustando dinamicamente os parâmetros da simulação[3]. No entanto, na maioria destes sistemas, os seus componentes (de simulação, monitorização, visualização e controlo) ficam, na maioria dos casos, fortemente dependentes do modelo computacional em causa, bem como da plataforma computacional sobre a qual o sistema foi desenvolvido. Durante os anos 1990, as dificuldades em lidar com a evolução deste tipo de ambientes foram sendo enfrentadas, com abordagens de desenvolvimento de *software* orientadas para componentes e com modelos para suportar a operação integrada[4] de ferramentas desenvolvidas separadamente.

Infraestrutura de suporte à monitorização. Dadas estas dificuldades, o objectivo desta tese foi o de procurar identificar um modelo definindo uma infraestrutura distribuída oferecendo uma base comum e genérica para suportar funcionalidades de monitorização, permitindo a interacção do utilizador ou das ferramentas de monitorização com as aplicações, conforme os requisitos específicos e os diferentes regimes de funcionamento adequados a cada utilização. É desejável que a infraestrutura que suporta o monitor seja o mais possível independente da plataforma computacional sobre a qual se executa, para que facilmente se possa dispor do monitor nas mais variadas arquitecturas. Por outro lado, é também conveniente conseguir a maior independência possível do monitor, relativamente aos modelos de programação ao nível da aplicação, permitindo, em alternativa, que se possa facilmente adaptar o sistema de monitorização, de modo a incorporar as ferramentas necessárias para cada cenário de utilização, bem como facilitar a extensão do sistema com novas funcionalidades para suportar novas situações.

Estes objectivos motivaram a definição de uma infraestrutura minimalista, flexível e adaptável, conforme as funcionalidades de monitorização pretendidas.

1.2 Enquadramento do trabalho

Os trabalhos que conduziram a esta dissertação decorreram no contexto de diversos projectos de investigação do Grupo de Processamento Paralelo e Distribuído do Departamento de Informática da FCT-UNL.

[3] Um processo designado por *computational steering*.
[4] *Interoperability.*

Alguns destes projectos foram de âmbito local ao departamento [99, 59] ou de cooperação com outros grupos da FCT-UNL [124]. Outros projectos foram de âmbito nacional [112, 116] ou relativos a contratos de investigação com companhias informáticas [108]. Outros projectos tiveram âmbito internacional, do programa Copernicus [23, 22, 28, 109] e do programa ESPRIT Working Group EuroTools da UE, ou envolveram cooperações bilaterais [39].

O autor desta dissertação esteve envolvido nestes projectos, em particular, nos diversos desenvolvimentos relativos ao suporte da monitorização de aplicações paralelas e distribuídas, em distintos contextos de utilização. A diversidade de cenários encontrados nestes projectos contribuiu para identificar as ideias apresentadas nesta tese e também para a sua validação experimental.

Nesta secção, apresentam-se, muito resumidamente, as principais dimensões investigadas nestes projectos, no que diz respeito à monitorização de aplicações paralelas e distribuídas.

1.2.1 Desenvolvimento de *software* para aplicações paralelas

Objectivos

Nos projectos SEPP, HPCTI, SEIHP e KIT, do programa Copernicus investigou-se um ambiente para o desenvolvimento de aplicações paralelas, centrado num modelo de programação paralela visual, com especificação das aplicações através num editor gráfico interactivo, a partir do qual se processava a geração automática de código, na linguagem C, estendida com as primitivas do modelo PVM - Parallel Virtual Machine [46]. O objectivo central era a exploração de conceitos de alto nível para o desenvolvimento das aplicações, de uma forma unificada, em todas as etapas do ciclo de desenvolvimento e integrando uma diversidade de ferramentas, desenvolvidas independentemente umas das outras: simulação para previsão e avaliação de desempenho, teste e depuração ao nível do modelo de programação visual (GRAPNEL [70]) e ao nível do C e PVM, suporte de estratégias de *mapping* e de equilíbrio da carga e análise do comportamento (simulado e real) dos programas, através de animação e visualização.

O projecto evidenciou as múltiplas dificuldades práticas de integração de ferramentas,

dada a sua heterogeneidade em termos de interfaces de utilização e de requisitos de plataformas de suporte específicas. Em particular, a interacção entre as ferramentas de monitorização, de simulação e de visualização baseou-se num formato de traço específico ao monitor utilizado no projecto (Tape/PVM [90, 40], efectuando-se, quando necessária, a conversão entre diferentes formatos de traços (por exemplo, o formato NPICL [129]), utilizados por outras ferramentas. A interacção sobretudo baseada em ficheiros de traço representou uma limitação ao grau de integração permitido entre as ferramentas.

A experiência obtida com estes projectos permitiu identificar as vantagens de uma infraestrutura de suporte à monitorização, que estabelecesse uma plataforma comum e suportasse a interacção com as múltiplas ferramentas do ambiente:

- para disponibilizar informação relevante às ferramentas de análise de desempenho, permitindo também a adaptação dinâmica das aplicações;
- para transmitir informação para efeitos de visualização do comportamento dos programas;
- para suportar abordagens combinadas de teste e depuração, nas actividades de recolha de dados e de controlo da execução dos programas.

A necessidade de suportar a interacção entre ferramentas concorrentes, a heterogeneidade destas ferramentas e o carácter dinâmico de tais ambientes, motivaram a definição do modelo *DAMS — Distributed Applications Monitoring System*, que se propõe nesta tese.

1.2.2 Monitorização para suportar a depuração distribuída

Nos projectos anteriormente mencionados, desenvolveu-se o depurador distribuído *DDBG* [27, 34]. Baseado numa arquitectura distribuída para suportar a depuração de programas C/PVM, oferecia uma interface (API) bem definida, facilitando assim a sua integração com outras ferramentas [77, 70, 38]. A arquitectura do *DDBG* consistia numa colecção de depuradores individuais (baseados no depurador *gdb*), distribuídos pelos múltiplos nós de uma arquitectura distribuída, cada depurador local ficando associado a um processo componente da aplicação. Um processo coordenador central era responsável pela interacção com o utilizador ou com cada ferramenta particular, e pelo encaminhamento dos pedidos e das respostas para os correspondentes nós.

Esta arquitectura exigia o apelo a funcionalidades básicas para a observação e o controlo de programas distribuídos, as quais nos pareceram, logo de início, poderem ser disponibilizadas por uma infraestrutura mais genérica de suporte à monitorização.

Assim, enquanto que, por um lado, uma linha de investigação separada desenvolvia o tema da depuração de programas [81, 85], por outro lado, os requisitos e as características do modelo *DAMS*, uma infraestrutura genérica de suporte à monitorização, eram identificados. Apesar de separadas, as duas linhas de desenvolvimento encontraram-se frequentemente, sempre que era conveniente ensaiar as funcionalidades de depuração sobre uma infraestrutura efectiva de monitorização [41].

1.2.3 Monitorização de distintos níveis de abstracção

A observação de computações paralelas e distribuídas a distintos níveis de abstracção é uma necessidade fundamental, conforme se discutiu anteriormente. O estudo das dimensões do modelo DAMS foi também conduzido face aos requisitos postos por distintos modelos de programação paralela e distribuída, situados em diferentes níveis de abstracção na hierarquia de camadas de um sistema computacional:

1. O modelo DAMS foi aplicado para o suporte à monitorização de programas paralelos na linguagem PVM-Prolog[93], permitindo comprovar a flexibilidade para observar diversas camadas de abstracção – desde o nível das construções de um programa em lógica, até ao nível das construções correspondentes da biblioteca C e interface do sistema PVM.

2. O modelo DAMS foi também aplicado para o suporte à monitorização de programas na linguagem C baseados no modelo MPI[102].

3. O modelo DAMS foi ainda aplicado para o suporte à monitorização de programas baseados no modelo Java[120].

Estes desenvolvimentos permitiram validar os conceitos do modelo DAMS, como intermediário entre as funcionalidades do nível de modelos de alto nível (Prolog, MPI, Java) e as camadas de mais baixo nível que suportam a observação e controlo dos componentes da aplicação.

1.2.4 Monitorização para suportar observação *online* e controlo dinâmico

Para além do suporte de ambientes integrados para o desenvolvimento de aplicações, visou-se também auxiliar o desenvolvimento de ambiente integrados para a gestão, observação e controlo de aplicações distribuídas. Sendo estas últimas, tipicamente, constituídas por componentes de simulação, visualização e controlo, parecia relevante analisar a aplicabilidade do modelo DAMS, também nestes contextos. Este estudo foi também conduzido no contexto de projectos em que o autor participou, relativamente ao desenvolvimento de um ambiente integrado para a resolução de problemas,[5] num domínio de aplicação específico para a execução paralela e distribuída, a visualização e o controlo dinâmico (*steering*) de algoritmos genéticos [30], em ambientes PVM sobre um *cluster Linux*. As necessidades colocadas ao modelo DAMS, nestes ambientes, diziam respeito ao suporte da configuração e gestão de recursos (incluindo a activação de componentes) da arquitectura de *software* distribuída, ao nível da aplicação, ao suporte da interacção dos elementos de computação com os de visualização e ao suporte dos componentes de controlo e suas interacções com o ambiente.

1.3 Contribuições

As principais contribuições deste trabalho são as seguintes:

- a proposta de uma abordagem, consubstanciada num modelo para a definição de arquitecturas de monitorização, satisfazendo os requisitos de flexibilidade e extensibilidade, para suportar cenários de observação e controlo de computações paralelas e distribuídas;

- a proposta de uma arquitectura modular que suporta uma infraestrutura com um núcleo minimalista e neutro em relação às funcionalidades específicas de monitorização e em relação às plataformas computacionais subjacentes; suportando a definição de um conjunto aberto de serviços específicos às necessidades de monitorização de cada caso;

[5] Habitualmente designado por *Problem-solving environment.*

- a realização de protótipos do modelo e sua infraestrutura de suporte, sobre duas plataformas computacionais distintas;

- o desenvolvimento de serviços e sua utilização em diversos cenários de observação (*online* e *offline*), de controlo (depuração de programas e controlo dinâmico de computações), a diversos níveis de abstracção.

1.4 Organização da dissertação

Este documento começa por caracterizar o domínio no qual se enquadra este trabalho e apresentar a motivação para o trabalho desenvolvido.

No capítulo 2, apresentam-se as dimensões da monitorização de computações paralelas e distribuídas, identificando os principais conceitos teóricos e as dificuldades intrínsecas à realização de monitores.

No capítulo 3, discutem-se os requisitos para a concepção de infraestruturas de monitorização flexíveis, extensíveis e adaptáveis, sobre as quais se possam realizar múltiplas funcionalidades de suporte à monitorização, dependentes das necessidades de cada cenário de utilização e das características de cada plataforma computacional de base.

No capítulo 4, apresentam-se os objectivos e as características do modelo geral para a infraestrutura que se propõe (*DAMS*), descrevendo uma arquitectura abstracta baseada num núcleo minimalista, sobre o qual se instalam serviços de monitorização específicos. Descrevem-se ainda as funcionalidades suportadas pelo núcleo e por dois serviços fundamentais, um de suporte à gestão de traços de execução e outro de suporte ao controlo dinâmico da execução.

No capítulo 5, discutem-se opções de implementação do núcleo, sobre diversas plataformas computacionais, em particular uma baseada no modelo de comunicação por mensagens *PVM* e outra baseada no modelo CORBA — *ORBit*.

No capítulo 6, são apresentados os diversos cenários de utilização que foram experimentalmente ensaiados para validar os conceitos e os mecanismos propostos pelo modelo DAMS e pela sua infraestrutura.

Finalmente, no capítulo 7, apresentam-se as conclusões, fazendo uma síntese dos prin-

cipais resultados obtidos com este trabalho e discutem-se possíveis direcções de trabalho futuro.

Capítulo 2

Fundamentos da Monitorização de Computações Distribuídas

Apresentamos neste capítulo o domínio no qual este trabalho se enquadra, e identifica-se o problema de observar ou controlar as aplicações paralelas e distribuídas e as principais abordagens ao seu tratamento.

2.1 Introdução

Num sentido lato, a actividade de monitorização envolve diversas tarefas, todas elas relacionadas com o tratamento de eventos: detecção, registo e processamento. A detecção de eventos consiste na observação de certas transições de estados nos processos sob monitorização e a correspondente análise de certas propriedades associadas aos estados observados, possivelmente seguida da execução de determinadas acções.

Dada a complexidade dos sistemas de computação paralela e distribuída, a concepção e realização de um sistema de monitorização deve ter em conta um compromisso entre o grau de precisão desejado e a perturbação introduzida pela observação. Para tal, devem considerar-se as seguintes dimensões principais:

1. um modelo formal que permita compreender as propriedades intrínsecas de um sis-

tema de monitorização, em particular, as que caracterizam as observações correctas (completas e adequadas);

2. a identificação das abstracções e dos mecanismos primitivos que devem ser incorporados num monitor, de modo a satisfazer os requisitos postos pelas ferramentas de desenvolvimento de programas e pelos sistemas de suporte à execução;

3. a identificação dos princípios arquitecturais que suportem a realização prática de sistemas de monitorização e uma base experimental que possibilite a sua validação, em cenários de utilização reais.

Neste capítulo, apresenta-se uma perspectiva daquelas dimensões, de um ponto de vista teórico e conceptual, deixando para o capítulo seguinte a discussão das principais abordagens para a concepção e realização de arquitecturas de suporte à monitorização.

2.2 Computações paralelas e distribuídas

As motivações para as aplicações paralelas e distribuídas podem ser classificadas segundo as seguintes principais dimensões, que identificam os objectivos que se pretendem atingir:

1. soluções que cumpram os níveis de desempenho exigidos pela especificação das aplicações;

2. graus de disponibilidade e confiabilidade que sejam adequados à natureza e especificação das aplicações;

3. soluções hardware/software que beneficiem da decomposição funcional de uma aplicação, isto é, baseada na especialização das funções atribuídas aos seus componentes individuais;

4. soluções em que a arquitectura hardware/software do sistema computacional se deva adaptar à distribuição geográfica da própria aplicação, seja em termos das entidades onde se processa a aquisição ou entrada de dados/comandos, seja em termos dos locais onde aqueles são arquivados, processados ou visualizados.

Como se vê, em qualquer dos casos acima, pode beneficiar-se da existência de arquitecturas hardware de múltiplos processadores:

1. Por exemplo, no primeiro caso, procuram-se desenvolver algoritmos para a resolução paralela dos problemas, explorando a sua decomposição em múltiplos processos concorrentes e a sua activação em processadores independentes. As principais preocupações, nestes casos, relacionam-se com a procura das melhores soluções para a partição dos problemas (em termos dos dados a processar e das funções a efectuar) e correspondente distribuição dos dados e processos pelos processadores disponíveis, com vista aos níveis desejados de desempenho. Diremos que, neste primeiro caso, o objectivo dominante das aplicações é o alto desempenho e o recurso à computação paralela é fortemente justificado.

2. No segundo caso, podem explorar-se múltiplas unidades físicas para fins de processamento ou de arquivo de dados, conforme forem determinadas pelos objectivos das aplicações ou pelas especificações dos serviços que devem ser disponibilizados. Diversos graus podem ser exigidos quanto à disponibilidade de um dado cenário aplicacional, incluindo a garantia de tolerância às falhas dos componentes hardware e software. O princípio da redundância ou da replicação, seja na dimensão espacial (replicar dados e/ ou operações em distintas unidades de arquivo ou de processamento), seja na dimensão temporal (duplicar/repetir a execução de operações, seja em simultâneo, seja em diferido, ou ainda, baseado na recuperação de estados do sistema, previamente armazenados).

3. No terceiro caso, a especialização dos componentes da arquitectura hardware/software pode ajudar no processo de decomposição funcional das aplicações, por razões de modularidade ou de eficiência.

4. No quarto caso, a repartição geográfica dos componentes da aplicação determina que a arquitectura do sistema distribuído seja concebida de forma a corresponder aos requisitos da aplicação. Se, nas aplicações distribuídas mais tradicionais (ou convencionais), as interacções estabelecidas são de natureza esporádica (envio de mensagens ou acesso a funcionalidades de serviços remotos), com esquemas de cooperação pouco “fortemente ligados”, com a melhoria dos débitos de transmissão das redes de comunicação, foram-se tornando possíveis formas de cooperação mais forte entre componentes geograficamente distribuídos. Por exemplo, em aplicações distribuídas que suportam

esquemas de trabalho cooperativo (CSCW), ou em aplicações de computação paralela que procuram explorar os modelos de "grid computing".

Esta caracterização leva-nos a designar as aplicações (ou os programas) em que estamos interessados neste trabalho, sob a designação genérica de *aplicações (ou programas) paralelas e distribuídas*, por vezes simplificando para *aplicações (ou programas) distribuídas*. Pretende-se, com isto, salientar que se tratam, por um lado, de aplicações nas quais se procura atingir o primeiro objectivo (*elevado desempenho* através da computação paralela e, por outro lado, que se consideram modelos de computação associados a arquitecturas distribuídas, para a execução dos programas.

Programas distribuídos. Neste trabalho, assume-se que um programa distribuído é composto por um conjunto de processos (possivelmente variável) que interagem entre si e cuja execução é suportada por um modelo de computação distribuída, ou seja, no qual se assume a ausência de um mecanismo de controlo centralizado, que fosse responsável pela coordenação global das acções individuais efectuadas pelos processos envolvidos na execução de um programa, sobre uma arquitectura de múltiplos nós. Estes nós são elementos com capacidades de execução, memória e comunicação (os quais podem ser "simples" unidades centrais de processamento (CPU) com acesso a memórias locais privadas, ou serem computadores autónomos, com os seus dispositivos periféricos dedicados e, possivelmente, com múltiplos processadores "locais"), interligados entre si por uma rede de comunicação.

Os processos que constituem o programa interagem entre si, sob diversas formas, estabelecendo relações de cooperação mais ou menos fortes, por exemplo, para a simples comunicação de mensagens ou para a resolução conjunta de um problema. Para além disso, a arquitectura de um sistema distribuído também pode suportar a execução de múltiplas aplicações independentes, partilhando os acessos aos recursos computacionais disponíveis nos nós que estejam integrados no sistema.

Diversos modelos de comunicação podem ser utilizados para a programação das interacções que os processos estabelecem ao nível das aplicações, quer baseados em abstracções de memória partilhada, quer de memória distribuída (por mensagens ou por invocação de funções remotas).

A execução de um programa distribuído envolve uma hierarquia de conceitos, correspondente aos diferentes níveis ou camadas de abstracção que a suportam:

- os *processos* nos quais o programa se decompõe logicamente e as interacções que estes estabelecem; os processos podem ser assimilados a nós de um grafo e as interacções entre eles podem ser representadas por arcos nesse grafo, obtendo-se assim uma representação lógica da aplicação;

- uma arquitectura distribuída disponibiliza, para suportar a execução dos processos, múltiplos elementos processadores virtuais; cada processador virtual pode ser realizado sobre os diversos tipos de nós de que se disponha em cada arquitectura, com o apoio de um sistema de suporte à execução e de operação adequado: seja sobre um monoprocessador, seja sobre um multiprocessador, ou ainda sobre um agregado de processadores (*cluster*) cujos elementos processadores podem comunicar entre si através de memória fisicamente partilhada ou por uma rede de interligação dedicada; os nós assumem-se interligados por uma rede física, suportando a comunicação por mensagens entre os processos, enquanto que os processadores virtuais se supõem ligados por uma rede lógica de canais de comunicação, que permite abstrair da topologia da infraestrutura de comunicação subjacente (incluindo as ligações físicas e o suporte providenciado pelo sistema de operação).

O nível dos *processos* descreve a organização lógica de uma aplicação. O nível dos *processadores virtuais* descreve uma arquitectura virtual intermédia que permite lidar com a diversidade dos tipos de nós e com a natureza heterogénea das arquitecturas dos sistemas distribuídos. A grande diversidade de arquitecturas virtuais e de sistemas distribuídos existentes apresenta variações em diversas dimensões:

- nos modelos computacionais associados à noção de processador virtual, para suportar diversos modelos dos processos ao nível da aplicação, por exemplo, com ou sem suporte de múltiplos fluxos de execução, com criação estática ou dinâmica de processos, com grupos de processos; nos diferentes paradigmas associados ao modelo de programação, seja, por exemplo, imperativo, lógico ou orientado aos objectos;

- nas diferentes formas como os processos interagem entre si, seja com base em modelos de memória partilhada ou de memória distribuída, isto é, baseados na troca de mensagens ou na invocação de pontos de acesso remotos (procedimentos, funções, métodos);

- nos diferentes grau de centralização / descentralização dos modelos de controlo da execução dos processos e da sua coordenação; por exemplo, com suporte para coordenação baseada em comportamentos de tipo mestre/escravo, ou com coordenação distribuída, baseada na cooperação simétrica entre entidades; ou ainda baseados em paradigmas específicos de distribuição dos dados e do controlo da execução — por exemplo, modelos de paralelismo a nível dos dados (*data parallel*) ou SPMD (*Single Program Multiple Data*);

- na homogeneidade / heterogeneidade do *hardware* e do *software* dos sistemas de suporte à execução dos múltiplos nós presentes no sistema.

Neste capítulo, assume-se que cada processo tem associado um único fluxo de controlo da execução (ou *thread*), sendo assim modelado por uma máquina de estados, sequencial. Um processo efectua transições autónomas de estados, correspondentes a eventos locais, e envolve-se em interacções com outros processos, que determinam transições de estados dos processos envolvidos, sujeitas a determinadas condições de sincronização.

Um programa distribuído[1] especifica determinados esquemas de decomposição de tarefas e de cooperação entre processos, baseando-se no modelo abstracto da linguagem de programação utilizada para exprimir a concorrência, a distribuição e a comunicação entre os processos. Assim, exibe um comportamento que pode ser descrito em termos das noções abstractas, por exemplo, processos e eventos, conforme são definidas pela semântica da linguagem utilizada.

Sistemas distribuídos. Um programa distribuído é executado no contexto de um sistema distribuído. Nesta dissertação, entende-se por sistema distribuído, um sistema que oferece os mecanismos computacionais (na mesma hipótese de ausência de um mecanismo central de controlo das acções dos processos) para suportar a execução de processos distribuídos, sobre uma arquitectura virtual distribuída definida segundo uma hierarquia de níveis, desde os intermédios, tais como as plataformas“middleware”, o sistema de operação e as camadas de protocolos de comunicação, até ao nível inferior das arquitecturas hardware dos computadores e das infraestruturas de ligação física.

[1] A designação de *programa concorrente* seria porventura mais adequada, neste ponto.

Ao executar um programa distribuído sobre um sistema distribuído, as entidades mais abstractas do programa (processos e eventos) vão corresponder[2] às entidades concretas, definidas pela arquitectura virtual que caracteriza o sistema distribuído. Habitualmente, ao nível do sistema distribuído, dispomos de um conceito de processo ao nível do sistema de operação (por exemplo, o conceito de "processo" ou o conceito de "*thread*" em Unix/Posix [107]) e de conceitos de base para a comunicação por mensagens (por exemplo, as portas de comunicação como os *sockets*). No entanto, como a arquitectura virtual que constitui o sistema distribuído é, em geral, ela própria, organizada segundo uma hierarquia de camadas, pode acontecer que as entidades mais concretas com que lidamos, ao nível do sistema distribuído, sejam, diferentes das do nível intermédio, por exemplo, objectos e mecanismos de invocação de métodos remotos (por exemplo, como em CORBA [106]), ou outras abstracções de nível intermédio, por exemplo, processos e mensagens, tal como nos sistemas PVM [46] ou MPI [102].

O reconhecimento dos diferentes níveis de abstracção em relação aos quais se podem descrever as acções desencadeadas pela execução de um programa distribuído, é uma das dimensões importantes para se compreenderem as dificuldades da monitorização, porque, para se compreender o comportamento de uma aplicação, irá ser necessário estabelecer a correspondência entre os sucessivos níveis de abstracção envolvidos na sua execução.

Características dos programas distribuídos. Diversas características tornam a observação do comportamento das computações (paralelas e) distribuídas mais difícil do que a das computações sequenciais. Nestas últimas, por existir um único processo, existe uma única sequência de eventos (e estados) a observar, segundo a linha temporal de execução do programa. Dispõe-se, também, de um mecanismo central de marcação das datas dos eventos, associado ao relógio físico do processador que executa o programa. Finalmente, por o sistema computacional de base ser um sistema centralizado, dispõe-se de um mecanismo global de observação (por exemplo, suportado pelo sistema de operação e baseado no relógio físico do processador) capaz de registar (e, depois, reconstruir) a sequência única de eventos produzida durante uma execução do programa. Tendo o programa sequencial, na grande maioria dos casos, um comportamento determinístico, é possível reproduzir os efeitos de uma execução anterior do programa, por exemplo, para efeitos de depuração de erros ou para apoiar outras análises sobre o comportamento do programa.

[2] Através de *mappings*.

Num programa distribuído, encontramos quatro principais dimensões relevantes para a problemática da observação de computações distribuídas:

- o programa pode envolver um grande número de processos, que são criados e destruídos dinamicamente, e estabelecem interacções ao longo da execução;
- o programa exibe um comportamento não-determinístico, devido à execução paralela e distribuída;
- não se dispõe, num sistema distribuído, de um mecanismo de observação global que permita a construção precisa, actualizada e consistente dos estados globais das computações e das sequências de eventos que ocorrem durante uma execução, em contraste com o que acontece com um sistema centralizado;
- a perturbação introduzida pelos mecanismos de observação, que se devem implantar junto de cada processo do programa para registar os eventos e permitir o seu tratamento, origina, ela própria, modificações no comportamento do programa, relativamente à sua execução livre, não sujeita a observação.

A primeira dimensão (*processos distribuídos dinâmicos e interactuantes*) exige a capacidade de fazer observações a dois níveis: ao nível global, para se compreenderem as interacções dos processos; ao nível local, para se analisar cada processo individualmente. Se, para além disso, a escala do programa, em termos de processos ou de processadores, atinge valores muito elevados (por exemplo, muitas dezenas ou centenas), então é necessário ter isso em conta, tanto ao nível das abstracções de observação oferecidas, como a nível da arquitectura do sistema de monitorização.

A segunda dimensão (*não-determinismo*) é uma consequência natural da concorrência exibida pelo programa, constituído por múltiplos processos e executados em múltiplos processadores. Torna-se, assim, possível explorar os benefícios do processamento paralelo, para reduzir o tempo de execução dos programas. No entanto, os processadores impõem diferentes ritmos de execução de instruções e as ligações entre eles exibem tempos de comunicação imprevisíveis e variáveis, características dos sistemas distribuídos *assíncronos*.

Ao observar o comportamento de um programa distribuído[3], distinguem-se dois tipos de não-determinismo:

[3]Na verdade, estas são características também presentes em programa concorrentes.

1. um não-determinismo que é externamente observável, sendo caracterizado em termos da relação entre as entradas (dados) e as saídas (resultados) do programa; nestes termos, um programa exibe um comportamento não-determinístico se, para os mesmos valores de entrada, pode produzir diferentes valores de saída, em execuções repetidas;

2. um não-determinismo interno e que resulta de os processos constituintes de um programa exibirem diferentes sequências de estados internos, possivelmente observáveis em execuções repetidas, em idênticas condições de entrada; este tipo de não-determinismo pode, ou não, originar, um comportamento não-determinístico externamente observável.

Um exemplo de um programa com não-determinismo interno, mas externamente determinístico, é o de um programa com múltiplos processos concorrentes, cujas acções sejam independentes entre si e que não acedam a recursos comuns. Em repetidas execuções desse programa, podem observar-se diferentes sequências de estados, assumidas internamente pelos processos, mas, nas mesmas condições de entrada, o programa produz os mesmos resultados finais, externamente observáveis.

O não-determinismo pode complicar a tentativa de observação de situações de erro num programa distribuído, por exemplo, quando múltiplas acções concorrentes, envolvendo processos distintos, estejam em conflito (isto é, a sua ordem de ocorrência seja relevante para assegurar a correcção do programa) e, no entanto, podem ocorrer em ordens diferentes, devido aos problemas das temporizações imprevisíveis e variáveis, acima mencionados. Estas situações podem ocorrer tanto em programas concorrentes baseados em comunicação por memória partilhada ou por memória distribuída. Estes problemas influenciaram a evolução dos modelos e ferramentas de apoio ao desenvolvimento de programas concorrentes, paralelos e distribuídos, a diversos níveis, procurando aumentar a confiança do programador na garantia da correcção dos programas: a detecção dessas situações; a avaliação de propriedades de correcção do programa que sejam válidas para quaisquer ordens de ocorrências de eventos; a concepção de técnicas para suportar a reprodução determinística das sequências de eventos verificados numa dada execução, de modo a ajudar a localização das causas dos erros no programa. Idênticas dificuldades podem ocorrer, quando o objectivo é a análise do comportamento dos programas, visando a avaliação do desempenho da sua execução, numa determinada arquitectura distribuída. Nestes casos, subtis alterações na ordem de execução das acções, sejam devidas à concorrência ou ao paralelismo, ou à variação na duração dos

intervalos de tempo associados, por exemplo, motivadas por variações imprevistas no número de processos ou no tráfego de mensagens na arquitectura distribuída, podem modificar radicalmente o comportamento de desempenho da execução do programa. A dificuldade da identificação destas situações é acrescida pelos efeitos combinados e pelas interferências entre mecanismos e estratégias aplicados em distintos níveis do sistema, por exemplo, suspensão da execução de um processo por falta de páginas em memória central, maiores tempos de espera por falta de linhas na *cache*, etc.

A terceira dimensão (*observação de estados globais*) é relevante porque, num sistema distribuído, só se conseguem observar os estados locais dos processos e as suas transições, através de mecanismos de observação remota, ou seja, com base em troca de mensagens entre o observador e o observado. Uma vez que o sistema distribuído não dispõe, por definição, de um mecanismo de controlo centralizado que permita capturar, a cada momento de uma computação, um estado global instantâneo, exacto e consistente (isto é, compatível com as relações de causalidade), um sistema de monitorização tem de suportar mecanismos para realizar a observação de estados globais. Na presença de um sistema distribuído assíncrono, isto é, em que cada processador tem um relógio local independente dos outros processadores e os tempos de transmissão de mensagens são imprevisíveis e variáveis, não se conhecendo um tempo limite superior fixo para o tempo de transmissão de mensagens, as observações permitidas são necessáriamente aproximadas, devendo, para cada cenário de aplicação, ser equacionadas as técnicas de observação mais adequadas.

A quarta dimensão (*perturbação devida à observação*) exige que o sistema de monitorização se baseie em mecanismos que provoquem a menor perturbação possível, tanto em termos de atraso no tempo de execução do programa como em termos de alterações induzidas na ordenação dos eventos, ou em novos eventos, produzidos por uma execução sob observação, quando comparados com os de uma execução livre, isto é, sem mecanismos de observação incorporados. Havendo sempre alguma perturbação, tornam-se necessários os estudos conducentes à avaliação do grau de perturbação efectivo, às tentativas para a sua redução e, eventualmente, o recurso a técnicas de compensação, que procurem construir observações nas quais se eliminem os efeitos das alterações introduzidas.

Computações distribuídas. De forma a podermos lidar com as dificuldades anteriormente apresentadas, é necessário dispor de conceitos que caracterizem formalmente as propriedades das computações distribuídas.

O conceito de computação distribuída representa todos os comportamentos que podem ser desencadeados pela execução de um programa distribuído num sistema distribuído. Uma execução do programa (*run*, na designação anglo-saxónica) corresponde a um desses casos possíveis.

O estado de um processo, num dado instante, é definido pelo conjunto dos valores das suas 'células de memória', incluindo os valores correntes das portas de comunicação com o exterior. Cada acção do processo modifica o seu estado, originando um evento. Duas principais classes de eventos são relevantes: os *eventos internos* representam transições locais de estado do processo, isto é, realizadas autonomamente, sem envolver outros processos; os *eventos externos ou de interacção* representam interacções entre os processos.

No modelo mais básico, ao nível das camadas do sistema de operação que suportam a realização da comunicação sobre a infraestrutura de comunicação do sistema distribuído, é habitual considerar dois tipos básicos de eventos de interacção, associados, respectivamente, ao envio e à recepção de mensagens. Formas de comunicação de mais alto nível definem outros tipos de eventos, podendo ser estabelecida a sua correspondência com os eventos básicos do nível do sistema distribuído. A compreensão das formas de realização daquela correspondência tem, frequentemente, um papel central na identificação das causas dos desvios ao comportamento desejado dos programas, relativamente à sua correcção, bem como ao seu desempenho.

Estando cada evento associado a uma transição de estado do processo, pode-se definir um processo P_i como uma sequência de eventos (também chamada a sua história local), numa ordenação total, em que o número de ordem atribuído a cada evento, se interpreta como uma data, marcada por um "relógio" lógico (um contador de eventos [78]), local ao processo.

A "história local" HL_i do processo P_i

$HL_i = e_i(0), e_i(1), e_i(2), ..., e_i(f)$

é uma ordenação total de eventos, desde o evento inicial, de criação, até ao evento final, de terminação (admitindo sequências finitas). Uma vez que, a cada evento, está associada uma transição entre dois estados sucessivos, esta sequência representa a evolução do processo, com todas as modificações das suas variáveis privadas e com todas as interacções do processo com o ambiente exterior. O evento $e_i(k)$ produz a transição para o estado $S_i(k)$. Assim, o evento inicial define o estado inicial $S_i(0)$ do processo e o evento $e_i(f)$ define o

estado final $S_i(f)$.

A união dos eventos de todas histórias locais de todos os processos envolvidos forma a "história global" (HG) da computação distribuída associada à execução do programa. Nem todas as permutações desses eventos correspondem a sequências de estados (ou ordenações de eventos) válidos, isto é, passíveis de ocorrer numa execução real do programa.

Só são válidas as ordenações de eventos que sejam compatíveis com a relação de precedência causal (denotada pelo símbolo $\rightarrow$) definida por Lamport [78].

A "computação distribuída" (CD), que descreve a execução de um programa por um sistema distribuído, é definida por um conjunto de eventos, parcialmente ordenado (*partially ordered set (poset)*[6]) pela relação de precedência causal:

$$CD = (HG, \rightarrow)$$

Intuitivamente, este conceito representa todas as possíveis combinações de eventos das histórias locais dos processos envolvidos num programa distribuído, sujeitas às restrições de causalidade que devem ser obedecidas em todas as possíveis execuções do programa por um sistema distribuído.

O aspecto essencial do processo de observação de computações distribuídas corresponde a registar e, posteriormente, 'reconstruir' a informação que descreva os estados globais da computação, bem como as sequências da sua evolução temporal.

Um "estado global" (EG) de uma computação distribuída, definida por um programa com n processos, é um n-tuplo de estados locais dos processos envolvidos:

$$EG = (EL_1, EL_2, ..., EL_n)$$

em que EL_i é o "estado local" do processo P_i, correspondente a algum prefixo da história local desse processo. O estado global inicial da computação, $EG(0)$, é definido pelo n-tuplo dos estados locais iniciais de todos os processos, e o estado global final $EG(f)$ é definido pelo n-tuplo dos estados finais de todos os processos. Partindo do estado inicial até se atingir o estado final, existem múltiplas combinações possíveis de estados globais intermédios, correspondentes a todas as permutações de estados locais que satisfazem a relação de precedência causal.

Para caracterizar a observação de estados globais válidos, define-se a noção de *corte* da computação distribuída. Um corte (*C*) é um subconjunto da história global, que representa uma história parcial da computação, construída à custa de um conjunto de prefixos de histórias locais, atingidos até um certo conjunto de eventos, um em cada processo. A "fronteira do corte" é o *n*-tuplo formado pelos últimos eventos no prefixo de cada história local, representando, assim, um "ponto de progresso", atingido em cada processo, numa dada execução. Estes conceitos podem ser ilustrados graficamente, num diagrama espaço-tempo, que representa as linhas de evolução temporais de cada processo.

De entre os cortes que se podem definir, só os cortes consistentes são relevantes para a observação ser considerada válida. Um corte *C* diz-se *consistente* se, para todos os eventos *e*, incluídos em *C*, todos os eventos *e'* tais que $e' \rightarrow e$ tem-se *e'* também, necessariamente, incluído em *C*. Um corte não é consistente se incluir algum evento *e*, e excluir algum dos eventos *e'* que precedam causalmente *e*. Intuitivamente, um corte consistente representa um ponto de observação global/colectivo válido, isto é, correspondente a um estado global da computação que poderia, efectivamente, ocorrer nalguma execução distribuída do programa.

2.3 Observação de estados globais de computações distribuídas

A noção de estado global de uma computação distribuída corresponde, intuitivamente, a um ponto de observação da colecção de estados locais de todos os processos (incluindo os estados dos seus canais de comunicação), tal como pode ser apercebido por um observador ideal, externo à computação sob observação. Num sistema distribuído, a forma habitual pela qual um observador externo pode construir tal visão do estado global, assenta em trocas de mensagens com cada processo remoto individual. Há, assim, diversos aspectos a ter em conta:

- O estado global observado pode ser *obsoleto*, quando a visão global acima mencionada é construída pelo observador da computação. Isto ocorre, por exemplo, se a observação é feita em simultâneo com a execução distribuída do programa (*online*). Isto é particularmente relevante no caso da monitorização de aplicações distribuídas, nas quais se queira modelar um comportamento reactivo, tornando-se particularmente

crítico nos sistemas distribuídos ditos de operação *em tempo real*. Idêntica situação ocorre nos casos em que a monitorização deve suportar a observação de uma computação (por exemplo, uma simulação distribuída) que deva ser controlada (*steered*) dinamicamente. Se a construção dos estados globais se faz apenas após terminada a execução do programa (*post-mortem ou offline*), para posterior análise e processamento, este problema não se coloca.

- O estado global construído pelo observador externo deve corresponder a um corte consistente da computação. Numa observação efectuada em simultâneo com a execução, podem assim observar-se os pontos de progresso até ao momento da observação. Numa observação *post-mortem* pode, teoricamente, aceder-se a diversos cortes, correspondentes a estados globais intermédios, desde o inicial ao final, desde que as ferramentas de monitorização o possibilitem. Sendo a monitorização de um programa distribuído baseada num observador externo que deve coligir a informação sobre os estados locais dos processos, a ocorrência de diferentes ordenações que violem a causalidade, na entrega das mensagens que contêm essa informação, pode originar a construção, pelo observador, de cortes inconsistentes. Num sistema distribuído, é preciso, por isso, aplicar mecanismos de entrega das mensagens ao observador, que preservem a causalidade [6].

- Em ambientes de desenvolvimento de programas, torna-se cada vez mais necessário recorrer a múltiplas ferramentas de apoio, por exemplo, para depuração, visualização, controlo interactivo, as quais desempenham papéis de observadores ou controladores da execução distribuída. Como essas ferramentas devem poder operar concorrentemente sobre os processos de um mesmo programa distribuído, há que oferecer garantias de que as observações realizadas pelas múltiplas ferramentas sejam consistentes entre si. Isto coloca a dificuldade de garantir que distintos observadores construam idênticas sequências de eventos, mesmo na presença de um sistema distribuído assíncrono. As soluções para este problema dependem dos cenários concretos de aplicação, que determinam quais os tipos de interacções, que se devem considerar, entre as ferramentas (por exemplo, directas ou indirectas, envolvendo apenas observação ou também envolvendo controlo).

2.4 Monitorização de computações paralelas e distribuídas

2.4.1 Abordagens de monitorização

Os métodos utilizados para a observação de computações distribuídas dependem dos requisitos postos ao sistema de monitorização, para cada cenário particular de aplicação, distinguindo-se dois principais métodos:

- *Offline ou post-mortem:* neste caso é possível analisar histórias globais completas que foram previamente registadas durante uma execução do programa ou até que foram obtidas por simulação, baseada num modelo do comportamento do programa.

- *Online ou em simultâneo:* neste caso é necessário ir construindo os estados globais consistentes, durante a própria execução do programa. Assim, este método lida com histórias globais parciais.

Do ponto de vista do modo de operação do sistema de monitorização, consideram-se duas principais abordagens [6]. Ambas se baseiam num observador externo, que se ”sobrepõe” ao sistema que se pretende observar e a que chamaremos um *monitor*.

A primeira abordagem baseia-se numa estratégia de monitorização *activa*, que procura construir um estado global da computação (*global snapshot*), por iniciativa explícita do monitor, que envia, para o conjunto de processos a observar, um pedido de informação e, ao receber as respostas, constrói um corte consistente. Esta abordagem tem a limitação de apenas permitir a observação de alguns cortes, podendo perder outros. Assim, não é adequada para a observação de propriedades da computação distribuída que sejam *instáveis*, isto é, que variem ao longo da execução. É, contudo, uma abordagem adequada para a análise ou detecção de propriedades *estáveis*, isto é, que, uma vez atingidas num certo estado global da computação, se mantêm verdadeiras até ao estado final (por exemplo, uma situação de *deadlock*). Também é um abordagem utilizada em situações em que o sistema de monitorização suporta pedidos de informação, por iniciativa do utilizador, sobre o estado corrente dos recursos computacionais (contextos de execução e de comunicação dos processos, bem como o estado dos processadores e dos componentes de arquivo e de comunicação existentes numa dada configuração da arquitectura do sistema distribuído). Estes pedidos podem

ser emitidos, quando necessário, por iniciativa do utilizador, ou periodicamente, por iniciativa de componentes do sistema de monitorização, que realizam amostragens do estado das computações e do sistema que as suporta, nos diversos nós da arquitectura distribuída, para suportar a análise (e eventual controlo) do comportamento das aplicações.

A segunda abordagem baseia-se numa estratégia de monitorização *passiva* que permite ao monitor ir construindo uma sequência de estados globais, desde o inicial até ao final. Para esse efeito, cada processo sob observação deve enviar uma mensagem ao monitor, por cada evento relevante detectado localmente. Compete ao monitor ir construindo os estados globais das computações, assim observados.

Ambas as abordagens têm utilização prática importante.

Monitorização activa. A monitorização activa é necessária para responder às necessidades de sistemas nos quais um utilizador ou uma ferramenta tomam a iniciativa de interrogar, de forma interactiva, o estado de uma computação em curso, apenas em momentos por eles determinados. Tal pode visar apenas a inspecção da evolução da computação, por exemplo, para efeitos de visualização, ou pode estar associada a um sistema de controlo dinâmico do comportamento da computação, baseado num ciclo de monitorização, decisão e controlo (designado habitualmente por *computational steering*). Este tipo de utilização tem-se tornado cada vez mais importante, na última década, com o desenvolvimento crescente de ambientes computacionais de suporte à simulação de processos complexos em variados domínios das Ciências e Engenharias. Por outro lado, no contexto da depuração de erros do programa, esta abordagem só é, em geral, eficaz e viável, se for complementada por outros métodos (por exemplo, de análise estática e de geração de cenários de teste) que identifiquem as ”regiões” e/ou os processos que interessam inspeccionar activamente, a cada momento, com base em ferramentas e ambientes de suporte à depuração interactiva [77].

Monitorização passiva. A monitorização passiva é interessante em ambientes nos quais seja mais conveniente ir coligindo informação sobre os estados sucessivos alcançados pela computação, sem que o utilizador ou uma ferramenta tenham de tomar alguma iniciativa para esse efeito. Em sistemas que permitem uma interpretação dos dados coligidos, feita *a posteriori*, por exemplo, para efeitos de avaliação da correcção do comportamento ou do desempenho de um programa, a abordagem de monitorização passiva é também frequente-

mente a escolhida. Uma das razões é o facto de nos dar diversos graus de liberdade quanto à especificação dos eventos relevantes, cuja ocorrência se pretenda monitorizar, bem como aos momentos em que se devem processar as tarefas de transporte, processamento e arquivo dos registos dos eventos sujeitos a monitorização. Estes graus de liberdade são importantes para nos ajudarem a satisfazer diversos compromissos de realização da arquitectura de suporte à monitorização, em particular, para reduzir os efeitos da perturbação introduzida pelo monitor, bem como com as formas de compensar tais efeitos.

Em particular, uma das dificuldades práticas da abordagem de monitorização passiva resulta de poder implicar o registo de enormes quantidades de dados, nos casos em que estes dados devam ser arquivados para análise *post-mortem*. Este problema tem motivado alternativas, baseadas na intervenção do utilizador que, interactivamente possa ir determinando quais os eventos (e o nível de abstracção) relevantes a cada momento do processo de desenvolvimento das aplicações. Neste contexto, combina-se a monitorização passiva com interfaces que permitem a intervenção do utilizador, bem como a visualização dos dados coligidos, evitando assim a necessidade de os armazenar (mas não resolvendo os problemas do seu transporte durante a execução). Se o objectivo é a depuração de erros, esta abordagem é também importante para apoiar o registo de traços da execução, baseados apenas em certos tipos de eventos relevantes, que venham posteriormente a ser utilizados para guiar a execução repetida do programa, de forma determinística, possibilitando re-análises de regiões ”suspeitas”. Nestes casos, a ideia de coligir a mínima informação possível numa primeira execução, registando-a num traço, para depois suportar análises mais detalhadas, permite também reduzir substancialmente a quantidade de dados a armazenar durante a execução e abre o caminho para formas mais adequadas, conduzidas pelo utilizador, quando se inspecciona o comportamento em modo de re-execução.

2.4.2 Funcionalidades para a monitorização de programas distribuídos

Os sistemas de monitorização suportam, conforme os cenários de utilização pretendidos, diversas actividades relacionadas, quer com o ciclo de desenvolvimento de programas, quer com o controlo dinâmico da sua execução. As funcionalidades que suportam a monitorização de programas distribuídos podem ser organizadas em torno de um ciclo básico que inclui os seguintes passos principais:

1. recolha dos dados relevantes sobre o comportamento dos programas;

2. tratamento dos dados recolhidos;

3. apresentação dos dados ao utilizador ou a outras ferramentas

Recolha de dados

Idealmente seria desejável dispor de informação de alto nível, especificada pelo utilizador, para indicar os tipos de dados relevantes, de forma a que o sistema de monitorização pudesse determinar, o mais automaticamente possível, os mecanismos adequados para a recolha dos dados, nas diversas camadas da hierarquia do sistema computacional que suporta a execução da aplicação. Essa especificação de alto nível ajudaria, por um lado, a focar a atenção na informação desejada pelo utilizador e, por outro lado, permitiria ao sistema, eventualmente, reduzir a sobrecarga computacional do processo de observação, por exemplo, reduzindo a quantidade dos dados a coligir.

A determinação dos tipos de dados relevantes e dos indicadores de comportamento que interessam ao utilizador, dependem dos objectivos de cada sistema. Por exemplo, se o objectivo é a avaliação do desempenho, um primeiro indicador global é o tempo total de execução, possivelmente depois decomposto em indicadores de tempos parciais, associados a distintas secções do programa e correspondentes perfis de comportamento. Se tais indicadores são os mais frequentes em programas sequenciais, eles são, contudo, insuficientes para ajudar na identificação das causas de problemas em programas distribuídos, nos quais é necessário analisar o seu comportamento em função das interacções entre processos (actividades de comunicação e de sincronização), bem como os efeitos das actividades de gestão do paralelismo (criação, eliminação e comutação de contextos dos processos), das estratégias de escalonamento dos processadores e de distribuição dos dados e das tarefas pelos múltiplos nós de uma arquitectura distribuída. Por outro lado, se para um utilizador ao nível da aplicação, os indicadores relevantes se referem ao comportamento desejado da aplicação, para um administrador de sistemas, os indicadores interessantes centrar-se-ão mais na eficiente utilização dos recursos computacionais.

As técnicas de recolha de dados em sistemas de monitorização podem ser classificadas em duas grandes categorias:

- *Baseadas na amostragem periódica de dados.* Esta técnica utiliza o princípio da monitorização activa dos estados de processos (ou processadores) individuais, permitindo calcular indicadores de comportamento, por exemplo, os intervalos de tempo associados à execução de rotinas ou o número de vezes em que são invocadas. Frequentemente utilizada para apoiar análises *post-mortem* e efectuada implicitamente por ferramentas que obtêm dados sobre o perfil da execução sequencial (tais como *prof*/*gprof* [50]), com recurso ao sistema de suporte à execução/sistema de operação, esta técnica tem aplicação limitada para programas distribuídos porque, por um lado, apenas obtém indicadores globais de comportamento (e não, por exemplo os comportamentos de interacção) e, por outro lado, porque está dependente do período de amostragem (em geral fixo ao nível do sistema de operação), o que pode inviabilizar a observação de certos comportamentos (como é característico da abordagem de monitorização activa).

- *Baseadas em eventos.* Esta técnica é mais geral e flexível, seguindo os princípios da monitorização passiva. A execução dos processos, ao desencadear transições de estados das computações, determina o registo de informação associada. Esta técnica pode ser aplicada à medição de intervalos de tempo associados à execução de secções arbitrárias de código, que são delimitadas por *sensores* que registam o início e o fim da execução. Pode também ser aplicada à contagem do número de ocorrências de eventos, obtendo indicadores globais de comportamento. Na sua forma mais geral, a recolha de dados baseada em eventos permite a especificação dos tipos de eventos relevantes e a informação a registar para cada tipo de evento. Ao registar a informação recolhida num traço, guardado em memória ou em ficheiro, esta técnica permite posteriores análises, com o nível de detalhe desejado em cada caso.

Abordagens para a instrumentação. Exceptuando os casos em que ferramentas tratam de perfilar, implicitamente, a execução dos programas, os sistemas de monitorização baseiam-se em técnicas de instrumentação explícita dos programas. A instrumentação consiste na inclusão, no programa original, de secções de código (*sensores ou actuadores*...)[4], que designaremos por *pontos de instrumentação*, cuja função é assinalar a ocorrência de eventos de interesse.

A instrumentação pode ser realizada inteiramente *por software*, sendo os pontos de ins-

[4]Também denominadas *probes.*

trumentação inseridos, quer manualmente, pelo próprio programador, em regiões arbitrárias do programa, quer automaticamente, pelo compilador ou pelo sistema de suporte à execução (por exemplo, em bibliotecas instrumentadas para fins específicos). A inserção dos pontos de instrumentação pode ocorrer a diversos níveis ou camadas do sistema computacional: directamente no código fonte do programa; no código fonte de bibliotecas específicas (ditas de instrumentação); no código objecto do programa, gerado pelo compilador; ou fazer parte do código do sistema de operação.

Em geral, as grandes vantagens da instrumentação *por software* são a flexibilidade na identificação de eventos ao nível de abstracção da própria aplicação, a facilidade com que se pode adaptar consoante as necessidades de observação em cada momento e a "portabilidade" das ferramentas, que não dependem de *hardware* específico. Estas vantagens são contrariadas pelo maior risco de perturbação no comportamento do programa face ao original, podendo originar alterações na ordenação de eventos e nas suas temporizações, as quais violem as especificações de comportamento correcto da aplicação. Esta perturbação resulta, por um lado, do aumento do tempo de execução do programa, devido ao código dos pontos de instrumentação e, por outro lado, das sobrecargas do processamento dos eventos. Este efeito é agravado porque, na instrumentação *por software*, os recursos computacionais da plataforma de suporte à execução (processadores, unidades de memória, canais físicos de comunicação) são partilhados pela aplicação e pelos componentes do sistema de monitorização.

Na tentativa de reduzir o grau de perturbação introduzida, identificam-se abordagens alternativas para a instrumentação:

- Instrumentação *por hardware*: o sistema de monitorização utiliza recursos computacionais dedicados, não partilhados com a aplicação. A detecção de eventos efectua-se a muito baixo nível, pois é desencadeada por dispositivos *hardware* (por exemplo, por inspecção de padrões de sinais num *bus*), obtendo-se assim uma perturbação mínima. As desvantagens desta abordagem são a sua dependência de *hardware específico* e o baixo nível de abstracção dos eventos detectados, tornando difícil a sua correlação com a semântica dos modelos de programação de mais alto nível.

- Instrumentação *híbrida*: comporta-se, por um lado, como a instrumentação *por hardware*, pois depende de hardware dedicado para detectar e processar os eventos; por outro lado, partilha com a instrumentação *por software* o aspecto de o programa a

monitorizar ser modificado de modo a executar o código dos pontos de instrumentação que activa os componentes hardware. A informação sobre os eventos detectados é transmitida, por canais *hardware* dedicados, para ser processada pelo monitor, que se executa também em *hardware* dedicado

Instrumentação estática ou dinâmica. A instrumentação estática apenas permite a inserção de pontos de instrumentação durante o desenvolvimento do programa, antes da execução. A instrumentação dinâmica exige uma implementação mais complexa, que suporte a inserção dos pontos de instrumentação, durante a execução do programa, mas permite explorar o comportamento dinâmico das aplicações. Torna-se, assim, possível decidir sobre a inserção de pontos instrumentação no programa, só quando tal é considerado necessário, durante a execução [96], podendo assim contribuir para uma redução da perturbação.

Os compromissos. A propósito das alternativas para a recolha de dados, podemos concluir que se trata sempre de procurar pontos de compromisso ou de equilíbrio entre a natureza da aplicação, o tipo de informação pretendida e as restrições impostas pelas plataformas computacionais disponíveis. Por um lado, procura-se uma observação com níveis de detalhe e de abstracção adequados e, por outro lado, procura-se reduzir a perturbação induzida na aplicação. O ponto (ou o intervalo) de equilíbrio corresponde a assegurar a correcção da observação, mesmo na presença dos efeitos da perturbação (este conceito é exposto mais detalhadamente na secção 2.5). Infelizmente não há uma resposta única e universal, competindo-nos a difícil tarefa de, dada a diversidade de graus de liberdade acima mencionados, procurarmos tomar as opções adequadas a cada caso. Para isto, é necessário proceder a intensa experimentação, exercitando e avaliando o impacto das diversas alternativas. Idealmente, seria desejável dispor de uma infraestrutura de suporte à monitorização flexível, que facilitasse o ensaio dessas alternativas. Esta é uma das motivações desta dissertação.

Tratamento dos dados

Os dados recolhidos são em geral sujeitos a tratamentos, para os tornar mais inteligíveis e compactos, por exemplo, reduzindo a sua quantidade por forma a facilitar a sua interpretação, aplicando filtros, calculando indicadores relevantes de desempenho e procurando estabelecer correlações entre os dados relativos a diferentes níveis de abstracção, por forma a facilitar a

compreensão dos efeitos combinados desses níveis sobre o comportamento global das computações. Por exemplo, para além da instrumentação dinâmica, há diversas abordagens para reduzir dinamicamente a quantidade dos dados:

- em certos casos é possível identificar conjuntos de processos (ou processadores) cujo comportamento seja considerado como representativo do comportamento global da aplicação, procedendo-se então a uma agregação de sumários de dados estatísticos;

- noutros casos, podem utilizar-se médias ou fórmulas que representem padrões temporais correspondentes a longas sequências de valores, permitindo reduzir o tamanho do traço registado, mas obrigando a um novo processamento adicional quando se pretende reconstruir os dados assim representados;

- noutros casos pode-se alterar dinamicamente o grau de pormenor da informação que se colige nos registos dos eventos, alternando entre as técnicas de traçar e de contar eventos [115];

A expressividade e flexibilidade das abordagens de especificação dos eventos e dos seus traços podem contribuir para facilitar o tratamento dos dados coligidos e a sua apresentação a distintas ferramentas, visando disponibilizar ao utilizador as mais adequadas perspectivas sobre o comportamento dos programas.

Classificação de eventos. Um evento elementar representa uma transição de estados, como se discute com maior pormenor na secção 2.5. Para além dos eventos elementares, podem considerar-se eventos *compostos* ou *estruturados*, os quais visam a facilitar a expressão de predicados globais, envolvendo condições dependentes dos estados de múltiplos processos distribuídos, permitindo assim, idealmente, facilitar a verificação do comportamento global das computações distribuídas. No entanto, a sua detecção, baseada na avaliação dos predicadoes globais é, em geral, muito complexa, por exemplo, a sua duração não é sequer conhecida, em sistemas distribuídos assíncronos. Apesar disso, formas simplificadas de eventos compostos e predicados globais têm sido estudadas, por exemplo, no contexto da depuração de programas [6]. Outro critério para a classificação de eventos é baseado na sua relação com as acções que provocam a transição de estados, por exemplo, o fluxo de controlo da execução de um processo, a alteração no valor de uma estrutura de dados ou as interacções entre

processos, por mensagens ou por memória partilhada. Finalmente, segundo o nível de abstracção, podemos distinguir eventos ao nível da aplicação, ao nível do sistema de operação ou ao nível da arquitectura hardware.

Atributos dos eventos. Os atributos de um evento dependem do seu nível de abstracção. Idealmente, um sistema de monitorização flexível deveria suportar uma especificação "aberta" para os formatos dos registos dos eventos. Isto não é incompatível com a inclusão de atributos habitualmente considerados mínimos de um evento, tais como o seu tipo, as localizações física e lógica associadas à sua detecção (o nome de um nó ou de um processo, por exemplo), e uma data (física ou lógica) associada à ocorrência do evento. Conforme o tipo do evento, outros atributos ser-lhe-ão associados. Por exemplo, os atributos de um evento associado à comunicação entre processos identificarão os participantes na comunicação e, possivelmente, o conteúdo da informação trocada.

Formatos dos traços de eventos. O formato de um traço define a sintaxe e a semântica associadas à interpretação dos registos dos eventos coligidos. A sua definição depende de um compromisso entre diversas dimensões:

- Flexibilidade na interpretação dos registos por múltiplas ferramentas;
- Eficiência do formato, em termos do tempo de execução e do espaço de memória que a sua manipulação exige.

No passado recente, têm-se verificado diversos esforços no sentido de definir formatos de traços normalizados, por forma a facilitar a interacção entre ferramentas desenvolvidas separadamente. O facto de a maioria dos sistemas de computação paralela e distribuída estarem em constante evolução, ao nível dos modelos, das plataformas computacionais e das próprias arquitecturas hardware, tem tornado difícil a definição consensual de formatos "universais" e "genéricos". Assim, existem muitos formatos que são dependentes de modelos e plataformas específicos, em relação aos quais a única possibilidade é recorrer a procedimentos de conversão, que possibilitem a cooperação (ainda que limitada) entre ferramentas. Há, contudo, abordagens que se destacam pela flexibilidade que permitem, baseadas em formatos baseados nas noções de orientação por objectos; ou em meta-formatos, que incluem as regras para a sua própria definição e interpretação dos seus registos [4, 115].

Armazenamento local e transporte de eventos. Num sistema de monitorização de programas distribuídos, a recolha de dados é efectuada localmente, em cada nó da arquitectura, junto de cada processo alvo. Habitualmente um nó central desempenha funções de coordenação global e trata do arquivo persistente ou da apresentação dos eventos aos utilizadores ou a outras ferramentas. Os registos de eventos coligidos em cada nó do sistema distribuído são, em geral, temporariamente armazenados nesse nó e, depois, transportados para o nó central, em momentos que dependem do modo de observação: *offline* ou *online*. No primeiro caso, os eventos apenas são transmitidos após o fim da execução da aplicação, excepto se, eventualmente, a capacidade de memória dos *buffers* locais for atingida, devendo então proceder-se ao seu envio para o nó central[5]. No segundo caso, os eventos vão sendo transmitidos durante a execução, seja para visualização do comportamento das computações, seja para suportar a sua análise durante a execução e possíveis acções de controlo dinâmico.

Apresentação dos dados

A apresentação dos dados ao utilizador ou a outras ferramentas permite posteriores análises e processamento: para a sua visualização, facilitando a compreensão do comportamento das aplicações paralelas e distribuídas [55, 71]; para apoiar a decisão em sistemas de controlo dinâmico (adaptável) [53]; e como fonte de informação para ferramentas de avaliação de desempenho que permitam identificar causas de problemas que devam ser corrigidos em iterações sucessivas do desenvolvimento dos programas [54].

Conclusão

Nos ambientes tradicionais de desenvolvimento, uma vez desenvolvida uma primeira versão de um programa, o seu comportamento pode ser observado, durante a execução, e a informação relevante pode ser coligida (com base nas técnicas acima discutidas) e analisada, em geral *a posteriori*, após o fim da execução, conduzindo à identificação de possíveis problemas e suas causas, originando sucessivas iterações que conduzirão a versões "melhoradas" do programa.

Tais ambientes pressupõem, em geral, configurações relativamente estáveis da plataforma computacional de suporte à execução, bem como aplicações relativamente estáticas

[5] Ou gravá-los em ficheiros locais ao nó.

(em termos, por exemplo, dos conjuntos de processos). Muitas das aplicações paralelas e distribuídas dos nossos dias exibem, por um lado, um comportamento dinâmico (em termos de variação dos seus componentes, bem como dos padrões de interacção que estabelecem entre si) e, por outro lado, são executadas sobre plataformas computacionais cada vez mais baseadas em sistemas heterogéneos e dinâmicos, com características de desempenho variáveis e imprevisíveis.

A necessidade de responder aos novos requisitos colocados por estas aplicações e ambientes computacionais, sugere novas perspectivas para o ciclo de actividades mencionado, tornando a observação *post-mortem* menos eficaz e exigindo mecanismos de observação e controlo dinâmico das aplicações, durante a execução. Do ponto de vista do suporte à monitorização, coloca-se a questão de como conciliar as exigências destes modos de funcionamento distintos.

Uma vez revistas, sucintamente, de um ponto de vista prático, as principais dimensões da monitorização de programas distribuídos, apresenta-se, na secção seguinte, uma perspectiva teórica, visando clarificar os conceitos fundamentais de um sistema de monitorização.

2.5 Modelo teórico de um sistema de monitorização

Nesta secção, revêem-se os conceitos teóricos mais específicos, que nos podem ajudar na compreensão dos problemas a resolver, relativamente à concepção das abstracções e à realização de uma arquitectura de suporte à monitorização.

Por forma a ajudar na compreensão de como um sistema de monitorização consegue reagir e actuar em resposta aos eventos relevantes, Marinescu et al [91] propuseram um modelo teórico baseado no paradigma *evento – acção*. Nesta secção, utilizamos este modelo para rever as principais características de um sistema de monitorização, permitindo-nos, assim, enquadrar a discussão do modelo proposto nesta tese, nos capítulos seguintes.

2.5.1 Introdução

A monitorização tem com objectivo a detecção das ocorrências de determinados eventos, durante a execução de um programa, por forma a obter informação que não se poderia

obter apenas pela análise do texto do programa ou por métodos de simulação. À detecção de um evento, segue-se um processamento e uma eventual reacção, que pode ou não exercer alguma forma de controlo sobre a própria computação sujeita a monitorização. Assim, a problemática da monitorização está intrinsecamente relacionada com a dos sistemas reactivos, conduzidos por eventos (*event driven*), sendo frequentemente apresentada em termos do seguinte ciclo básico [91]:

ocorrência de evento; avaliação de condição; execução de acção

As definições dos tipos de eventos, das condições a avaliar e das acções a efectuar, dependem dos objectivos da monitorização e da natureza das aplicações em causa. Como se viu anteriormente, a monitorização pode servir os mais diversos objectivos, desde o apoio à actividade de depuração de erros ou à avaliação do desempenho dos programas, até ao apoio a sistemas de controlo dinâmico, seja para fins de re-equilíbrio da carga de trabalhos (*load balancing*), para fins de tolerância a falhas, ou para fins de controlo dinâmico (adaptável) das computações.

Um modelo teórico permite identificar os aspectos fundamentais que caracterizam um sistema de monitorização e que são comuns aos diversos tipos de sistemas.

2.5.2 Sistema de monitorização

Um *Sistema de monitorização S* é um sistema composto por um (sub)sistema alvo *A* (a aplicação a monitorizar) e um (sub)sistema monitor *M* (responsável pela realização do ciclo básico de monitorização). O sistema *S* é definido (por [91]) pelo seguinte triplo:

$S = (A, M, Cam)$

em que:

$A = (Pa, Ca)$ é o sistema *Alvo*, formado pelo conjunto de processos *Pa*, que compõem o programa distribuído sujeito a monitorização, e o conjunto *Ca*, dos canais de comunicação que interligam os processos em *Pa*.

$M = (Pm, Cm)$ é o sistema *Monitor*, formado pelo conjunto de processos *Pm* que realizam as acções de monitorização, e o conjunto *Cm* dos canais de comunicação que interligam os processos em *Pm*.

Cam é o conjunto dos canais de comunicação que interligam processos em *Pa* e processos em *Pm*.

Em é o conjunto dos eventos, gerados pela execução de *A*, que são reconhecíveis por *M*.

Os conjuntos *Pa* e *Pm* são disjuntos , bem como os conjuntos *Ca*, *Cm* e *Cam*.

A figura 2.1 ilustra esta organização de um sistema de monitorização.

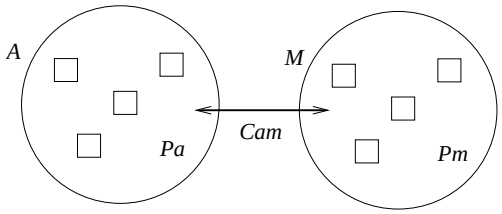

Figura 2.1: Organização de um sistema de monitorização

Processos. Um processo (*p*) em *Pa* é definido por uma sequência de estados, cujas transições, desde o estado inicial, definem uma sequência de eventos que, como se discutiu em secção anterior, caracterizam a história ou o comportamento observável do processo. Os processos em *Pa* podem ter canais de comunicação (de entrada ou de saída) que lhes estão associados.

O estado de um canal de comunicação é definido pelo conjunto das mensagens em trânsito, ou seja, que foram enviadas para o canal mas ainda não foram entregues através do canal (ou seja, ainda não consumidas).

O estado (*s*) de um processo em *Pa* é definido pelo valor de todas as suas variáveis e dos seus canais de comunicação, dependendo do nível de abstracção no qual se descreve o processo. Por exemplo, se o processo é definido ao nível do processador físico, o seu estado é caracterizado pelos valores das células de memória e dos registadores do processador. Se o processo for definido, por exemplo, ao nível da máquina abstracta associada a uma linguagem de programação, o seu estado será caracterizado em termos das construções definidas pela semântica operacional dessa linguagem.

Eventos. Um evento (*e*) num processo (*p*) em *Pa* é uma entidade atómica (com duração nula) que reflecte uma transição de um estado de p ou de um canal associado a *p*. Um evento é definido pelo quíntuplo seguinte:

$e = (p, s, s', c, msg)$

em que:

p é o processo no qual o evento ocorre, s é o estado de p que precede imediatamente a ocorrência de e, s' é o estado de p que se segue imediatamente à ocorrência de e. No estado s' inclui-se o estado do canal c que pode ser modificado pela ocorrência do evento, no caso de este envolver a transmissão de uma mensagem *msg*.

Monitor. O comportamento de cada processo (*Pm*) em *M* pode ser modelado como um ciclo eterno:

```
while TRUE do
 {
  AGUARDAR um dos eventos em definidos em Em;
  EXECUTAR uma das acções ami definidas em Acm,
                                  associada ao evento em;
 }
```

em que *em* é um dos tipos de eventos pertencentes ao conjunto *Em* dos eventos que são reconhecidos pelo monitor e am_i é uma das acções do conjunto *Acm* de acções, que o monitor é capaz de executar, em resposta à ocorrência dos eventos de *Em*.

2.5.3 Perturbação devida à observação

O resultado esperado da monitorização de uma computação distribuída é a geração de traço(s) que represente(m) correctamente a ordenação dos eventos e a informação associada a estes. A noção de correcção, neste contexto, significa que, por um lado, todos os eventos de interesse são registados no traçado (uma noção de *completude* da observação) e, por outro lado, que apenas aqueles eventos são observados e a sua ordenação reflecte a mesma ordenação, tal como seria observada se o subsistema alvo *A* fosse executado sem monitorização, isto é, nas condições $S = A$ (uma noção de "adequação" ou "fidelidade").

Um monitor *intrusivo* modifica a ordenação dos eventos observados, ou a sua temporização, ou provoca novas transições de estados no sistema alvo, relativamente às execuções de *A* efectuadas sem monitorização.

Conforme a natureza de cada aplicação e os objectivos do sistema de monitorização, um determinado grau de intrusão pode ser tolerável, mesmo não sendo nulo (o que é impossível de assegurar). Por exemplo, em aplicações criticamente dependentes de restrições temporais (em sistemas reactivos dito de controlo em tempo real), a intrusão pode originar, não apenas modificações na ordenação observada dos eventos, mas também violações das metas temporais (*deadlines*) que caracterizam o comportamento correcto da aplicação. Noutros casos, mesmo que sem as restrições críticas de tempo real, a perturbação pode afectar a ordem dos eventos, originando ou mascarando erros devidos ao não-determinismo da execução concorrente dos processos do programa alvo. Nestas situações, o teste e a depuração dos programas, bem como a sua observação para efeitos de avaliação do desempenho, ficam enormemente dificultados, exigindo técnicas específicas para facilitar o desenvolvimento dos programas.

Noutros casos, a perturbação pode limitar-se a aumentar muito o tempo total de execução do programa alvo, devido às sobrecargas computacionais causadas pela execução dos processos do sistema monitor, mas garantir, apesar disso, a completude e a adequação das observações.

Causas lógicas da intrusão

A intrusão pode ser devida ao próprio modelo de observação do sistema monitor (sec. 2.4), relativamente à detecção dos eventos e aos tipos de acções por eles desencadeadas. Por exemplo, um depurador interactivo cujo modelo de detecção de eventos seja baseado na colocação de pontos de paragem da execução do programa alvo (*breakpoints*) tem um elevado grau de intrusão associado à detecção de eventos. De igual modo, um monitor que, na sequência da detecção de um evento, desencadeia a suspensão da execução de processos do programa alvo, é altamente intrusivo nessas acções.

A intrusão devida à detecção dos eventos produzidos pela execução do sistema alvo depende do modo de realização dessa detecção pela plataforma computacional (hardware/software) que serve de base ao sistema de monitorização.

A intrusão devida às acções (*Acm*) desencadeadas pelo sistema monitor está, no modelo apresentado, dependente do sentido do fluxo da informação nos canais *Cam*, que ligam o sistema alvo *A* e o sistema monitor *M*. Se estes canais são exclusivamente unidireccionais, no sentido de *A* para *M*, então as acções do monitor não podem ser intrusivas, porque o mo-

nitor não tem forma de actuar sobre os processos do sistema alvo. Deve notar-se aqui que, no modelo apresentado, os conjuntos de canais *Ca*, *Cm* e *Cam* são disjuntos. Pelo contrário, se existem ligações bidireccionais entre *A* e *M*, mesmo que apenas devidas a canais implícitos de *M* para *A*, que veiculem sinais de confirmação (*acknowledgement*) e determinem a sincronização de processos em *A* com processos em *M*, abre-se a possibilidade para acções intrusivas. Por exemplo, se, na detecção de um evento, a execução de um processo de *A* ficar suspensa até que o monitor confirme a recepção da notificação do evento, há lugar para a possível modificação na ordenação e temporização dos eventos observados. Note-se que esta situação pode ser indirectamente causada pela escassez ou pela partilha de recursos computacionais, por exemplo, se o sistema de monitorização for obrigado a suspender a execução de um processo *Pa* até que o monitor consiga registar alguma informação num *buffer*, se este for limitado, ou se tal depender de alguma sincronização no acesso concorrente ao *buffer*, se este for partilhado.

Causas arquitecturais da intrusão

Nas realizações práticas de um sistemas de monitorização, há que estabelecer a correspondência entre os conceitos do modelo (*Pa, Ca, Pm, Cm e Cam, Em, Acm*) e a arquitectura do sistema de monitorização, bem como da realização desta com base nos recursos computacionais de uma determinada plataforma de suporte à execução de programas (constituída pelas bibliotecas de suporte a uma linguagem, pelo sistema de operação e pela arquitectura hardware subjacente). Em termos das entidades do modelo, as seguintes condições devem ser satisfeitas para que um monitor seja não intrusivo [91]:

- os processos de *A* e de *M* não podem ser realizados sobre recursos computacionais partilhados, para além dos que suportam os canais de monitorização *Cam*;
- os canais *Cam* devem ser unidireccionais, no sentido de *A* para *M*.

A situação ideal, para um grau de intrusão mínimo, exige suporte hardware separado para a realização de *A* e de *M*. Em particular, os processos *Pm* em *M* deveriam ser executados por processadores dedicados e as suas comunicações internas deveriam ser suportadas por ligações físicas dedicadas. No entanto, na prática, a abordagem de monitorização a adoptar em cada caso depende criticamente do cenário de aplicação, em particular determinando

a identificação do tipo de eventos relevantes e do nível de abstracção no qual se situam, o que origina uma diversidade de soluções, com distinto impacto no grau de perturbação introduzida.

Para se compreenderem as dificuldades e os compromissos que se enfrentam para a realização da situação ideal, há que discutir em pormenor as possíveis concepções da arquitectura do sistema de monitorização (capítulo 3).

2.5.4 A dimensão temporal e a completude da observação

Uma vez especificados os tipos de eventos relevantes, que definem o conjunto *Em*, a execução do programa alvo *A* desencadeia um processo de natureza dinâmica, ao longo do qual se pretende registar a história global das computações definidas pelo programa alvo. Para cada evento *e* pertencente a *Em*, associado a uma transição de estado de um processo *Pa* em *A*, o sistema monitor *M*, efectua a seguinte sequência de passos:

1. reconhecimento do evento, correspondente à recepção, em *M*, de alguma notificação da ocorrência de *e* (por exemplo, a detecção de um sinal num bus hardware ou a recepção de uma mensagem, enviada por *Pa*, através de um dos canais de *Cam*, para algum processo *Pm* de *M*);
2. início de uma acção am_i do conjunto *Acm* de acções definidas em *M*;
3. terminação da acção am_i associada ao processamento do evento.

Para cada evento em *Em*, o sistema de monitorização *M* exibe diversos parâmetros que o caracterizam em função dos tempos associados aos passos acima apresentados:

- (*lr*) latência do reconhecimento do evento: definida pelo intervalo de tempo que decorre entre a ocorrência do evento em *A* e o seu reconhecimento por *M*;
- (*ld*) latência do disparo da acção de processamento do evento, definida pelo intervalo de tempo entre o reconhecimento do evento em *M* e o início da execução da acção *acm*;
- (*dp*) duração do processamento, definida pelo intervalo de tempo entre o início e a terminação da acção *acm*;

A soma dos três parâmetros anteriores define o tempo total (*trp*) de reacção e processamento desse evento pelo sistema *M*.

Por outro lado, para cada evento *e* que ocorre numa execução de um processo *Pa* em *A*, existe um tempo de permanência (*ts*) associado ao estado *s*, que tenha sido alcançado em consequência da ocorrência de *e*. Este tempo é definido pelo intervalo de tempo durante o qual o processo *Pa* se mantém no estado *s*, até que decorra a próxima transição de estados em *Pa*.

Um sistema *S* diz-se *completamente observável* se todos os eventos (do conjunto *Em*) gerados por execuções dos processos *Pa*, são reconhecidos e processados por *M*. Para isto se verificar, é necessário que o sistema *M* disponha de tempo suficiente para reconhecer e processar cada evento *e1* gerado por cada processo *Pa*, antes de ocorrer o próximo evento *e2* nesse processo.

Uma condição para *S* ser completamente observável é que o tempo (*trp*) de reconhecimento e processamento de qualquer evento em *Em* seja inferior ao tempo de permanência (*ts*) do estado *s*, alcançado pelo processo, devido a esse evento. Por um lado, em certos cenários (análise em *offline*) o processamento de cada evento, durante a execução, pode ser reduzido ao mínimo indispensável (por exemplo, registo de informação mínima, em nó local à ocorrência do evento), deixando para mais tarde formas de processamento mais complexas. Por outro lado, o monitor *M* pode dispor de mecanismos separados de reconhecimento e de processamento de eventos, bem como de múltiplos processos *Pm*, que processem, em paralelo, as sucessivas ocorrências dos eventos gerados por *Pa*. No entanto, a coordenação das acções de tal monitor poderá tornar-se demasiadamente complexa.

Num monitor *M* que possa induzir transições adicionais de estados nos processos *Pa* do sistema alvo *A*, consegue sempre garantir-se que o sistema é completamente observável, quanto mais não seja pela suspensão forçada da execução de processos de *Pa*, enquanto não se completar o processamento de cada evento. Veja-se, por exemplo, o caso de um depurador interactivo baseado em 'breakpoints'. O mesmo pode não ser, contudo, aceitável num sistema de controlo dinâmico baseado em 'steering' de uma simulação, no qual tipicamente se pretendem modificar parâmetros que afectem as computações em curso, mas garantindo uma perturbação mínima, para além da desejada pelo controlo adaptativo. A este respeito também se pode argumentar que nem sempre a exigência de ser completamente observável é necessária. Em certos cenários de aplicação de sistemas de monitorização, basta que o

sistema *S* seja *estatisticamente observável*, ou seja, garanta que o número médio de eventos perdidos (não reconhecidos/não processados) por *M*, seja limitado superiormente. Uma condição que assegura esta propriedade verifica-se se a taxa esperada de ocorrência de eventos por unidade de tempo, num processo *Pa*, for inferior à taxa esperada de eventos que, por unidade de tempo, podem ser processados por processos *Pm*.

2.5.5 A dimensão temporal e a adequação da observação

O esquema seguinte, ilustra o efeito da monitorização, comparando os traços da execução do sistema alvo *A* e do sistema *S* (*A* com monitorização *M*):

Sem monitorização:
$A \rightarrow exec \rightarrow Tr_A$
Com monitorização:
$A + M \rightarrow exec \rightarrow Tr_{A+M}$

em que Tr_A é o traço da execução livre do sistema *A* e Tr_{A+M} é o traço obtido da execução do sistema *S*. Uma observação é correcta (completa e adequada), conforme visto anteriormente, se o traço da execução do sistema *S* for equivalente ao traço da execução do sistema *A*. Um sistema *S* diz-se correcto se, para todas as possíveis execuções de *A* e de *A* com *M*, os correspondentes traços são equivalentes.

Na definição mais estrita, a equivalência entre dois traços T_1 e T_2 exige que as seguintes condições sejam satisfeitas:

1. os dois traços contêm os mesmos conjuntos de eventos;
2. os eventos causalmente relacionados entre si, figuram em ambos os traços, pela mesma ordem.

A condição 1 determina que a monitorização regista única e exclusivamente os eventos que correspondem às transições de estados que são verificadas na execução dos processos do sistema alvo *A*. Esta exigência pode ser relaxada, permitindo que o traço Tr_{A+M} inclua novos eventos, para além dos que correspondem à execução livre, desde que a inclusão dos

eventos não afecte as condições de completude e adequação apresentadas em 2.5.3, quando aplicadas aos eventos comuns a A e em Tr_{A+M}.

A condição 2 determina que a monitorização não introduza alterações à ordenação dos eventos que sejam causalmente dependentes entre si.

Ordenação de eventos em sistemas de monitorização. As abordagens para garantir observações globais em que a ordenação dos eventos preserve a causalidade, no contexto dos sistemas de monitorização, são inspiradas em abordagens semelhantes [6], desenvolvidas no contexto dos sistemas distribuídos, os quais não dispõem de uma referência temporal única e global a todos os nós da arquitectura do sistema. Nestes últimos, há duas abordagens principais [121]: uma baseia-se em algoritmos que procuram a sincronização dos múltiplos relógios físicos existentes nos diversos nós da arquitectura distribuída, enquanto que a outra abordagem se baseia no conceito de relógio lógico [78] ou de relógio vectorial [94]. No entanto, no caso da monitorização de aplicações paralelas e distribuídas, a solução para este problema depende da abordagem de observação adoptada:

1. se a observação dos eventos é apresentada ao utilizador (ou a ferramentas de análise) em modo *post-mortem*, é possível, com base na informação coligida no traço de execução, reconstruir as dependências entre eventos e produzir um traço cuja ordenação de eventos reflecte a causalidade [90];

2. se a observação, pelo utilizador, pelo contrário, se deve efectuar durante a execução, devem então aplicar-se as técnicas habituais dos sistemas distribuídos, acima mencionadas.

No âmbito dos trabalhos desta dissertação, a perspectiva assumida, relativamente a este problema foi a admitir que, para cada cenário de aplicação, se deverá adoptar a abordagem de monitorização mais adequada, incluindo a solução para o problema da ordenação de eventos.

2.6 Conclusões

Neste capítulo, discutiram-se os conceitos de computação paralela e distribuída, reviram-se as funcionalidades e dimensões de um sistema de monitorização e discutiu-se um modelo

genérico de um sistema de monitorização. O objectivo desta exposição foi o de procurar clarificar a natureza das acções desencadeadas pelo monitor e o impacto das diversas abordagens alternativas, na correcção (completude e adequação) das observações.

A aplicação prática dos conceitos discutidos neste capítulo torna-se difícil, por duas razões. Por um lado, devido à diversidade de abordagens de suporte à monitorização e à sua dependência de cada caso concreto. Por outro lado, devido à complexidade da avaliação e do cálculo dos parâmetros e tempos definidos no modelo, quer para o sistema alvo, quer para o sistema monitor, pois que tais parâmetros dependem criticamente do tipo de computações geradas pela execução do programa alvo, bem como das características e tempos de resposta exibidos pela realização da arquitectura de monitorização sobre uma dada plataforma computacional distribuída.

A primeira dificuldade permite explicar a razão pela qual muitos dos sistemas de monitorização existentes têm objectivos dedicados a modelos de aplicações e plataformas específicas. A preocupação com o desenvolvimento de ambientes integrados e flexíveis, que suportem o desenvolvimento de programas, em cenários de aplicação variáveis, tem sido uma tendência recente. A segunda dificuldade tem inspirado investigação recente, sobretudo no que diz respeito à problemática da previsão e avaliação do desempenho, mas o seu estudo no contexto mais geral dos diversos tipos de observação e controlo de aplicações distribuídas é ainda um tema em aberto [54].

Capítulo 3

Arquitecturas de Suporte à Monitorização

Neste capítulo, introduzem-se os requisitos para a concepção de infraestruturas de monitorização, e apresentam-se as várias arquitecturas abstractas de suporte à monitorização, dependentes das necessidades de cada cenário de utilização e das características de cada plataforma computacional de base.

3.1 Arquitectura do sistema de monitorização

A arquitectura de um sistema de monitorização é baseada num conjunto de componentes *hardware*/*software*, cuja organização e modelos de execução permitem realizar a correspondência entre os conceitos teóricos definidos por um modelo de um sistema de monitorização e uma plataforma computacional de base.

Nesta dissertação, a arquitectura do sistema de monitorização é designada por *infraestrutura de monitorização* e o conjunto das camadas subjacentes é designado por *plataforma computacional de base* (Fig. 3.1).

Na camada designada por *plataforma computacional de base* (nível 2 da Fig. 3.1) incluem-se as funcionalidades habitualmente oferecidas por camadas *middleware* e pelos sistemas de operação, que suportam a execução paralela e distribuída. Por exemplo, no contexto dos trabalhos experimentais desenvolvidos nesta dissertação, consideraram-se, a este nível,

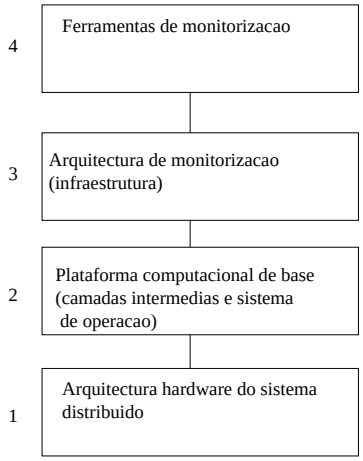

Figura 3.1: Hierarquia de camadas de suporte ao monitor.

dois ambientes distintos: num deles, a plataforma considerada foi suportada pelo sistema PVM [46] e sistemas de operação Linux e OSF/1; no outro, essa plataforma foi suportada pelo sistema ORBit/CORBA [106] sobre o sistema de operação Linux.

Em geral, as camadas 3 e 4, devem incluir todas as funcionalidades de suporte à monitorização, associadas ao ciclo de detecção, reconhecimento e processamento dos eventos. No nível 4 vamos considerar como ferramentas, as aplicações ou outros sistemas consumidores da informação recolhida, para os quais a monitorização é efectuada. Na camada de nível 3, designada por *infraestrutura de monitorização*, consideramos incluídas as funcionalidades básicas que suportam a monitorização, interagindo com a aplicação alvo ou sua plataforma computacional, com vista à recolha de informação, ou ao seu controlo.

3.2 Requisitos para a concepção de arquitecturas de monitorização

3.2.1 Introdução

Dadas as dificuldades para o desenvolvimento de aplicações paralelas e distribuídas, reconhece-se a importância dos mecanismos de apoio à observação do comportamento das computações. Por um lado, as ferramentas de monitorização complementam as funcionalidades de outras ferramentas, por exemplo, as que suportam a simulação ou a avaliação do

desempenho, na medida em que a monitorização permite obter informação sobre a execução real das aplicações. Por outro lado, a monitorização é um aspecto central nos ambientes de aplicações distribuídas em cujas configurações ou componentes possam ocorrer alterações imprevisíveis, cuja detecção pelos monitores permite desencadear as reacções adequadas, por exemplo, para efectuar acções de emergência, para equilibrar a carga de trabalho do sistema ou para assegurar adequados graus de tolerância a falhas.

Assim, não é de estranhar que, no passado recente, tenham sido desenvolvidos muitos sistemas e ferramentas de monitorização, habitualmente vocacionados para objectivos muito específicos, para além de serem, na sua maioria, fortemente dependentes de ambientes de programação específicos, bem como de sistemas de operação e arquitecturas *hardware* particulares.

Um número extremamente reduzido de tais sistemas ou ferramentas podem considerar-se passíveis de uma utilização mais genérica, mas mesmo esses sistemas não são fáceis de adaptar, para responder a diferentes requisitos de utilização ou a diferentes plataformas computacionais.

A crescente utilização de modelos de computação paralela e distribuída, que caracteriza as aplicações actuais, tem também contribuído para uma maior motivação, por parte da comunidade académica e industrial, para a concepção de ambientes de monitorização e controlo que satisfaçam a maior diversidade possível de requisitos. A seguinte lista ilustra tal diversidade, numa enumeração não exaustiva:

- avaliar o desempenho das aplicações;
- observar o comportamento das aplicações; e suportar a sua visualização, de forma *post-mortem* ou *online*;
- controlar a execução das aplicações, por exemplo, para fins de teste e depuração, por exemplo, para assegurar a re-execução determinística de aplicações;
- controlar a execução, para suportar ambientes de operação interactiva, em modo de *steering* computacional, nos quais um utilizador humano ou um agente exercem um controlo dinâmico sobre a evolução das computações, visando um comportamento adaptável;

- monitorizar a evolução das aplicações, visando a gestão equilibrada e eficiente dos recursos computacionais utilizados;
- suportar a reconfiguração das aplicações e dos sistemas computacionais subjacentes, por forma a assegurar disponibilidade e tolerância às falhas;
- suportar a reconfiguração dinâmica das aplicações por forma a reagir e adaptarem-se às modificações induzidas pelo utilizador ou pelas próprias computações em curso;
- suportar a construção e a composição de ambientes integrados para o desenvolvimento de *software* e para o suporte à colaboração entre utilizadores, baseados na composição de múltiplas ferramentas complementares, cuja execução concorrente e cooperação devem ser coordenadas.

3.2.2 Abordagens de realização da arquitectura

Face a estes cenários e à sua realização em ambientes de computação paralela e distribuída (heterogéneos, dinâmicos e com escala variável), não seria viável, nem sequer eficiente, realizar um sistema de apoio à monitorização monolítico, que abarcasse todas as utilizações acima mencionadas (e também outras, que eventualmente não constem daquela lista...). A escolha mais adequada será baseada numa abordagem de concepção modular, cujo princípio tem vindo, aliás, a ser aplicado em outros contextos, a diversos níveis de um sistema de computação. A este respeito, podemos mencionar, também, a título ilustrativo, o movimento que, na década de 1980, originou os desenvolvimentos dos micro-núcleos ao nível dos sistemas de operação dos computadores. Também no caso do suporte à monitorização se pode conceber a noção de um núcleo básico oferecendo um pequeno conjunto de funcionalidades, sobre as quais se possam realizar as funcionalidades requeridas pelos modos de utilização específicos.

Para a concepção do núcleo, torna-se necessário identificar um conjunto de funcionalidades 'mínimas' para um sistema de monitorização. Este conjunto deve ser:

- completo, no sentido de se poderem, sobre as suas primitivas básicas, realizar a diversidade de funcionalidades específicas ao nível das aplicações;
- genérico ou neutro, no sentido de não ser dependente de um ambiente de programação ou plataforma de computação particulares;

- eficiente e previsível: eficiente, no sentido de permitir realizações eficientes das suas funcionalidades, que tirem partido das plataformas e arquitecturas computacionais sobre as quais assenta; previsível, no sentido de cada uma das suas funcionalidades exibir um comportamento esperado 'proporcional' às características e complexidades das funções envolvidas na sua realização

- extensível: no sentido de permitir a inclusão de novas funcionalidades, sem exigir modificações radicais à arquitectura conceptual do sistema de monitorização.

É natural esperar que haja múltiplas respostas para a definição de um núcleo de suporte à monitorização. Mais adiante, são discutidas diversas propostas nesse sentido. Na proposta feita nesta tese, desenvolveu-se um modelo que procura cumprir os objectivos acima enunciados. Para a validação dum tal modelo, foi necessário verificar de que modo o núcleo possibilita diversos tipos de desenvolvimentos:

- especificação das diversas opções que configuram os modelos, modos de operação e interacções dos componentes correspondentes às diversas fases da monitorização de uma aplicação distribuída: detecção de eventos, processamento local, filtragem, transporte de eventos, processamento global, análise e visualização;

- especificação dos serviços requeridos para a observação e controlo das aplicações alvo, incluindo a sua configuração, recolha de dados ou outros serviços específicos;

- suporte da configuração e activação dos serviços acima mencionados;

- processamento dos comandos e interacções entre as interfaces de utilização, ao nível das ferramentas, e sua propagação até atingirem os componentes internos do sistema de monitorização, por forma a suportarem a consulta e o controlo do estado das computações;

- extensão incremental do ambiente de monitorização, através da inclusão de componentes que, assentes sobre o núcleo, suportem novas funcionalidades e a sua interacção com as ferramentas e interfaces de utilização que lhes estejam associadas; isto deve ser conseguido preservando a independência entre a interface de utilização ao nível da ferramenta e as características de mais baixo nível da arquitectura de monitorização e das plataformas computacionais subjacentes;

- composição modular de ferramentas elementares individuais, como forma de obter funcionalidades mais complexas;
- gestão e coordenação das interacções entre múltiplas ferramentas concorrentes, permitindo aos responsáveis pelo desenvolvimento de cada tipo de ambiente, implementar modos de acesso coordenado aos múltiplos componentes de uma aplicação alvo.

Neste trabalho o estudo e avaliação da "eficiência e previsibilidade" do sistema desenvolvido não foi efectuado, na medida em que se considerou prioritário o desenvolvimento dos conceitos e a sua validação experimental da sua funcionalidade face a distintos cenários de utilização.

3.2.3 Conclusão

Para compreender as opções de realização da arquitectura de monitorização nas secções seguintes deste capítulo, apresenta-se uma breve panorâmica da evolução dos sistemas de monitorização, centrada na identificação das interfaces, dos componentes e das suas interacções, que caracterizam diversas abordagens de concepção de arquitecturas de monitorização e que influenciam, a nosso ver, o modo como os objectivos, anteriormente enunciados, podem ser cumpridos.

3.3 Sistemas monolíticos de monitorização

As soluções "tradicionais" visavam objectivos e funcionalidades restritos e rigidamente incorporados na arquitectura do monitor, sendo em geral, fortemente dependentes do seu subsistema de suporte à execução (Fig. 3.2 *a*). Nestes sistemas, apenas se podia activar uma ferramenta de cada vez e não era possível ajustar o ambiente de monitorização a diferentes objectivos ou acrescentar-lhe novas funcionalidades.

3.3.1 Interacção baseada em ficheiros de traço

Uma abordagem simples e habitual para suportar a interacção entre diferentes ferramentas, baseia-se numa ferramenta dedicada à recolha de traços de execução, e as restantes

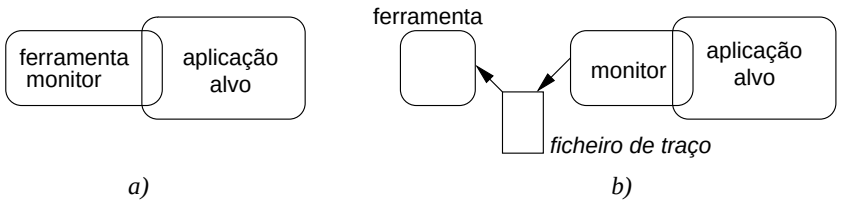

Figura 3.2: *a)* Sistema monolítico; *b)* Interface através de traço

utilizam estes traçados para analisarem o comportamento da aplicação, por exemplo, para monitorizar e avaliar o desempenho de aplicações. Há que acordar num formato aberto e normalizado para o traço, regulando as regras de definição e interpretação dos seus registos, para que se possam servir as mais variadas ferramentas (Fig. 3.2 *b*). Com estas soluções, conseguem-se, por exemplo, disponibilizar dados, previamente coligidos por um monitor, para processamento *post-mortem* por ferramentas de visualização ou de avaliação de desempenho. Novas ferramentas podem ser integradas em tais "ambientes de monitorização" desde que sejam concebidas de forma a serem compatíveis com os formatos acordados para os traços, ou se permitirem a aplicação de passos intermédios de conversão dos formatos.

A procura de definições normalizadas para os formatos dos registos de traço encontra aqui uma forte justificação e tem merecido os esforços das comunidades académica e industrial.

Um exemplo "típico" de um formato normalizado (*standard*) e um dos mais utilizados pela comunidade da computação paralela, é o formato utilizado pela biblioteca PICL *Portable Instrumented Communication Library* [129]. Este formato tem sido utilizado por múltiplos sistemas e ferramentas de monitorização, quer directamente, quer através de programas de conversão. Uma boa parte do seu êxito deve-se, muito possivelmente, à sua relação com a ferramenta de visualização de computações paralelas ParaGraph [55], bem como ao facto de a semântica da especificação do formato do traço ser relativamente neutra, face à diversidade de plataformas e sistemas subjacentes. A sua importância é também ilustrada pelo facto deste formato de traço ter sido estendido, numa nova versão, para se adequar ao sistema MPI [102], originando o formato designado por MPICL, que assenta numa interface, *standard* para o MPI, para instrumentar as aplicações, através de uma biblioteca de instrumentação baseada na norma *MPI Profiling Interface* [102].

Uma abordagem mais expressiva e flexível foi a seguida no projecto Pablo [115], quanto

à definição do formato SDDF – *Self Defined Data Format* [4]. Este segue a abordagem de definir um meta-formato com base no qual os vários registos de eventos podem ser definidos, com uma filosofia de concepção semelhante à seguida em outros contextos (por exemplo, a da norma XML [126]).

Associados aos ambientes de monitorização centrados em formatos específicos, têm-se, em geral, bibliotecas que disponibilizam funções básicas de leitura e escrita sobre os registos de eventos dos ficheiros de traço, segundo os formatos adequados (como os formatos usados no MPE/MPICH [17] e respectiva ferramenta Jumpshot), bem como outras funções que facilitem o processamento dos traços (p.e. compressão, conversão, etc.)

3.3.2 Interfaces para a instrumentação e controlo

Ao nível interno, os sistemas de monitorização recorrem a primitivas de baixo-nível, para suportar a instrumentação e o controlo dos elementos (processos, objectos, *threads*), que constituem uma aplicação distribuída. As funções que definem essas primitivas podem ser suportadas por distintas interfaces (API), em diferentes camadas ou níveis da plataforma computacional de base que suporta a execução: bibliotecas específicas, sistema de operação e a própria arquitectura *hardware*. São estas interfaces de baixo-nível que dão acesso aos dispositivos designados como *sensores* e *actuadores*.

A normalização dessas interfaces de baixo-nível contribui para facilitar a "portabilidade" da infraestrutura de monitorização e, portanto, das próprias ferramentas que a utilizam, na medida em que permite isolá-las das dependências, de nível mais baixo, da plataforma computacional subjacente.

Nestas abordagens, algumas funções de suporte à instrumentação e controlo são definidas separadamente da infraestrutura de monitorização e são oferecidas facilidades de instrumentação neutras, que podem ser adaptadas a diferentes sistemas de suporte à execução e a diferentes sistemas de operação (Fig. 3.3).

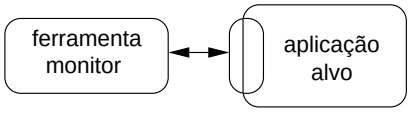

Figura 3.3: Interfaces para controlo e instrumentação

Como exemplos, podemos citar as iniciativas dos projectos PTOOLS – *Parallel Tools Consortium*, que visam a especificação normalizada das funções do nível mais baixo da infraestrutura de monitorização. Por exemplo, a interface PAPI – *Performance API* [12] dá acesso aos valores de contadores hardware, habitualmente suportados pelos CPU modernos. Também as rotinas PTR – *Portable Timing Routines* [114] permitem medir intervalos de tempo de execução dos programas, expressos em termos de tempos totais decorridos (*wall clock time*), de tempo de processamento no CPU, a nível do utilizador (*user CPU time*) ou do sistema (*system CPU time*).

Uma das propostas mais interessantes é a da biblioteca de instrumentação dinâmica DynInst – *Dynamic Instrumentation Library* [13], desenvolvida inicialmente no âmbito do projecto Paradyn [96]. A biblioteca suporta a instrumentação dinâmica do código executável de um programa, permitindo a inserção de código no programa, durante a sua execução. O ambiente desenvolvido permite ao utilizador, através de uma interface interactiva, navegar pelo texto do programa e instalar código de instrumentação (*code patches*) em pontos específicos. Tais secções de código são definidas com base numa descrição abstracta e depois convertidas no código executável, nativo de cada arquitectura *hardware* específica suportada pelo ambiente DynInst. Ainda que esta realização se baseie em funcionalidades do nível do sistema de operação, para inspeccionar e controlar processos, assim como para inserir dinamicamente nestes pedaços de código, como são suportadas múltiplas arquitecturas, permite-se uma razoável “portabilidade”.

As abordagens como as acima mencionadas, revelam os benefícios de se dispor de interfaces normalizadas que, ao nível da interacção com as funcionalidades de instrumentação de mais baixo nível, possam ser utilizadas por camadas de nível superior de um sistema de monitorização distribuída.

3.3.3 Interfaces e arquitecturas de suporte à monitorização

As abordagens baseadas em traços estão habitualmente associadas aos métodos de análise *post-mortem*, mas colocam diversas dificuldades para o suporte de cenários que exijam a interacção *online* com uma aplicação, durante a sua execução. Estes últimos cenários exigem interfaces e protocolos que regulem as interacções que devem ser estabelecidas entre as ferramentas e a aplicação, as bibliotecas de suporte, o sistema de operação e a própria

arquitectura *hardware* (Fig. 3.4). Modos de interacção síncrona e assíncrona devem ser suportados, e deve dispor-se de uma arquitectura de monitorização distribuída que facilite o acesso a componentes locais e remotos de uma forma transparente, segundo os principais requisitos mencionados em 3.2.

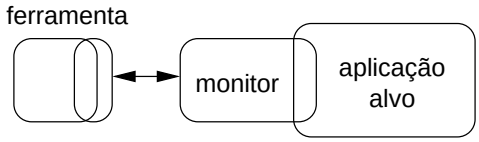

Figura 3.4: Separando a monitorização das ferramentas

DPCL. Uma abordagem interessante para melhorar a acessibilidade a interfaces de baixo-nível que suportam a instrumentação e o controlo, é baseada na biblioteca DPCL – *Dynamic Probe Class Library* [110]. Esta biblioteca define uma interface que se baseia no sistema DynInst para instrumentação dinâmica de processos. A arquitectura de DPCL assenta numa infraestrutura que utiliza o modelo cliente/servidor para permitir a uma ferramenta individual interactuar com múltiplos *Servidores DPCL*, um localizado em cada nó da arquitectura do sistema distribuído. Assim, permite-se que uma ferramenta tenha acesso e controle todos os processos de uma aplicação distribuída. Para além disso, cada servidor DPCL aceita interacções com múltiplos clientes concorrentes, permitindo assim que múltiplas ferramentas utilizem a infraestrutura simultaneamente, e disponibilizem, assim, observações ou modos de controlo complementares, para analisar ou controlar uma aplicação distribuída (Fig. 3.5). As dificuldades que estes modos de operação colocam, relativamente à resolução de eventuais conflitos e interferências entre as acções desencadeadas por aquelas ferramentas, não são tratadas pelo sistema DPCL.

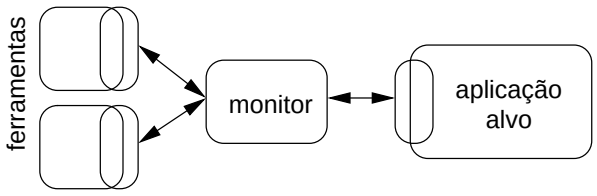

Figura 3.5: Infraestrutura de monitorização flexível

FIRST. Um outro exemplo de uma abordagem baseada em interfaces normalizadas para realizar uma infraestrutura de suporte à monitorização, é dada pelo sistema FIRST [111]. Este sistema visa, na sua essência, o suporte à realização de serviços de monitorização distribuída, utilizando a plataforma CORBA [106] como plataforma computacional de base e utilizando a biblioteca DynInst para o acesso local a cada processo de uma aplicação distribuída e as rotinas definidas pela interface PTR [114] para a medição de tempos. Uma interface gráfica permite ao utilizador, navegando pelo texto fonte de um programa, especificar pontos desejados de observação e controlo.

OMIS. O projecto OMIS – *Online Monitoring Interface Specificatoin* [88] representa um esforço sistemático no sentido de suportar uma interface de monitorização *standard*, assentando sobre uma infraestrutura aberta para o suporte ao desenvolvimento de ferramentas e de serviços de observação e controlo. A especificação OMIS define uma interface genérica para a inspecção e controlo de processos distribuídos e para suportar a interacção das ferramentas com o sistema de monitorização. O modelo de interacção é baseado no paradigma *evento* $\rightarrow$ *acção* e a interface permite a especificação, por cada ferramenta cliente do sistema, dos eventos em que tem interesse, de entre um conjunto de eventos pré-definidos (ver [88]) e as respectivas acções a desencadear, a quando da sua detecção. A arquitectura está pensada para suportar extensões, com novos eventos e acções. Múltiplas ferramentas concorrentes podem interagir com os processos de uma aplicação distribuída, recorrendo aos serviços da infraestrutura que suporta a interface OMIS. O sistema oferece um suporte limitado para a operação integrada dessas ferramentas.

As arquitecturas de monitorização ditas conformes com o modelo OMIS, podem basear-se em distintas plataformas computacionais de base: a primeira desenvolvida foi o sistema OCM-OMIS *OMIS Compliant Monitor*, baseado na plataforma PVM [46]. O projecto OMIS foi validado experimentalmente com base na adaptação de ferramentas de depuração e de visualização (DETOP e VISTOP [105]), [123], que haviam sido desenvolvidas pelo grupo de TUM em projectos anteriores, sobre diferentes plataformas de computação paralela.

Outros sistemas existentes. Dada a diversidade de sistemas de monitorização, nos mais diversos contextos de utilização, é impossível incluir aqui uma discussão mais detalhada todos os sistemas existentes. No entanto, múltiplos têm sido os esforços no desenvolvimento de novas ferramentas e sistemas de monitorização, assim como o reconhecimento da comu-

nidade na normalização de interfaces, harmonização de modelos, que têm levado à formação de consórcios para esses fins, assim como para a divulgação dos vários sistemas, como sejam o APART [2], EuroTools [44], PTools [114] e o NHSE [113].

3.4 A motivação para o desenvolvimento do modelo DAMS

3.4.1 Introdução

O modelo *DAMS* tem como principal objectivo promover uma abordagem de suporte à monitorização de computações paralelas e distribuídas, baseada numa arquitectura flexível que assenta num núcleo de base ao qual se sobrepõem serviços específicos. Cada serviço visa suportar uma determinada funcionalidade de monitorização e controlo. A motivação para este modelo resulta, conforme se discutiu em secções anteriores, do reconhecimento da diversidade de cenários de aplicação, plataformas computacionais e, portanto, da diversidade de abordagens e soluções para suportar a monitorização.

Segundo o modelo *DAMS*, em vez de incorporar um conjunto fixo de funcionalidades numa infraestrutura monolítica, exige-se que cada funcionalidade específica seja encapsulada como um serviço. Assim, o modelo *DAMS* promove a definição de uma colecção aberta de serviços, que recorrem a um conjunto mínimo de funcionalidades, oferecidas por uma infraestrutura ou núcleo de base. O objectivo do núcleo é o de servir de intermediário:

- entre os serviços e os processos da aplicação alvo;
- entre as ferramentas e os serviços disponíveis;

3.4.2 O desenvolvimento incremental do modelo *DAMS*

Dada a quantidade de dimensões envolvidas para atingir o objectivo de construir uma infraestrutura de suporte à monitorização que cumpra os requisitos anteriormente apontados, a abordagem seguida neste trabalho foi incremental e estreitamente relacionada com desenvolvimentos e ensaios experimentais que foram realizados no contexto de múltiplos projectos de investigação nos quais o autor esteve envolvido.

A origem e a evolução do modelo DAMS – *Distributed Applications Monitoring System*, uma infraestrutura de suporte à monitorização de aplicações paralelas e distribuídas –, foram fortemente condicionadas pelas necessidades, decorrentes desses projectos, de se dispor de uma infraestrutura distribuída flexível para suportar serviços de observação e de controlo de aplicações paralelas e distribuídas.

As diversas dimensões que foram exploradas através de desenvolvimentos experimentais permitiram ir validando as diversas versões do modelo DAMS, bem como fazê-lo evoluir, a partir de uma primeira realização de um protótipo simplificado, para uma nova versão do modelo e sua implementação.

3.5 Conclusões

Dos desenvolvimentos e experimentação anteriormente mencionados, emergiu a proposta do novo modelo *DAMS*. Este modelo, baseado numa noção de um núcleo básico, o qual pode ser estendido com conjuntos de serviços, conforme as necessidades de cada cenário aplicacional, representa, em nossa opinião, uma abordagem adequada para responder aos requisitos acima enunciados. Nos capítulos seguintes desta dissertação, procura-se evidenciar esta tese, descrevendo os conceitos do modelo, as abordagens de implementação e ilustrando a flexibilidade da sua utilização.

Capítulo 4

O Modelo DAMS

Neste capítulo introduzem-se os objectivos de um sistema para monitorização e controlo de aplicações distribuídas, e seus requisitos para responder aos problemas referidos no capítulo anterior. É proposto um modelo, denominado DAMS, e são apresentadas as suas interfaces abstractas.

4.1 Introdução

Para se obter um ambiente para a observação e controlo de aplicações distribuídas mais completo do que o oferecido por uma única ferramenta, recorre-se muitas vezes a um conjunto variado de ferramentas, cada uma com um objectivo específico. O utilizador recorre às várias funcionalidades oferecidas por cada ferramenta consoante o objectivo que pretende atingir. Seria possivelmente desejável recorrer à integração ou à cooperação entre várias ferramentas, tentando tirar partido do melhor que cada uma tem para oferecer. No entanto, estas ferramentas são normalmente concebidas independentemente, para um sistema ou fim específicos, e a cooperação entre estas é na maior parte dos casos diminuta, senão impossível. A disponibilidade dessas ferramentas depende em geral da plataforma que suporta a execução da própria aplicação. Mesmo a utilização simultânea de várias ferramentas pode levar à observação de estados incoerentes entre as várias ferramentas e a interferências que podem originar comportamentos erróneos na observação e no controlo da aplicação, que as ferramentas exercem.

Surge assim a necessidade de dispor de mecanismos que garantam a coerência nas observações e no controlo efectuado sobre a aplicação alvo, sobretudo quando se usam várias ferramentas complementares. Isto é particularmente importante e difícil quando o programa é executado sobre uma arquitectura distribuída. Esta dificuldade acrescida advém, por um lado, do carácter concorrente da sua execução, onde se torna quase impossível garantir que se observa correctamente a sequência de acções de uma execução do programa, nem se cobrem as possibilidades de todas as execuções desse programa; por outro lado, advém da complexidade que estas aplicações atingem em termos dos recursos usados, fluxos de execução concorrentes e suas interacções. As próprias ferramentas, para tratarem das dificuldades referidas, tornam-se mais complexas do que as equivalentes usadas para programas sequenciais (quando existem) assim como as suas necessidades em termos das infraestruturas que as suportam.

A existência de várias ferramentas, explorando diferentes abordagens e para os vários modelos de programação e suas plataformas de suporte à execução, tornam desejável utilizar novas ferramentas e poder combinar várias ferramentas com objectivos diferentes para melhor se conseguir o objectivo pretendido, procurando tirar partido das vantagens de cada uma. Tenta-se assim obter "ambientes" mais poderosos recorrendo à integração e composição de várias ferramentas que melhor se adequam ao objectivo de cada cenário de aplicação a monitorizar e/ou a controlar.

Também a existência de variadas plataformas *hardware* e *software* (modelos de programação, bibliotecas e sistemas de operação) tornam difícil a banalização deste tipo de ferramentas e de sistemas para controlo e observação, e ainda mais qualquer tipo de normalização que pudesse permitir um mais fácil transporte de ferramentas entre múltiplas plataformas. Procura-se então facilitar o desenvolvimento de novas ferramentas, assim como tornar mais fácil o seu transporte, e das ferramentas já existentes, entre as diversas plataformas *hardware*/*software*, pela existência de uma infraestrutura de suporte comum.

Revela-se de particular importância considerar os aspectos da coexistência, da coordenação e da sincronização entre as várias ferramentas, bem como destas relativamente à aplicação distribuída alvo, mas as soluções destes problemas dependem criticamente de cada cenário de utilização. A utilização destes tipos de ferramentas em vários ambientes, plataformas de suporte à execução da aplicação, ou de suporte à infraestrutura de monitorização, apontam para uma arquitectura de *software* por camadas que abstraiam os detalhes dessas

plataformas e facilitem a sua portabilidade para diferentes ambientes. Por outro lado, as funcionalidades comuns às várias ferramentas devem ser oferecidas pelo sistema de monitorização, permitindo a sua partilha e coordenação entre as várias ferramentas. Cada cenário de utilização definirá os requisitos dessa infraestrutura de monitorização ajustada a cada caso, poupando na utilização de recursos que em geral concorre com a da própria aplicação observada, permitindo a sua fácil adaptação, por adição ou remoção de funcionalidades, consoante cada situação.

4.2 Primeira abordagem

A primeira abordagem a esta problemática, no âmbito dos trabalhos realizados no Departamento de Informática da FCT/UNL, surgiu nos projectos SEPP [23] e HPCTI [22] do programa Copernicus, nos quais o autor participou. Nestes, identificou-se a necessidade de uma infraestrutura capaz de ser utilizada como suporte a diversas ferramentas que estavam sendo desenvolvidas para a depuração de programas distribuídos C/PVM. Uma primeira experiência onde se desenvolveu um sistema para controlo de depuradores distribuídos [34], motivou a investigação de uma arquitectura de suporte mais genérica e flexível, para poder responder aos diferentes requisitos das várias ferramentas.

A primeira instância desse conceito, a DAMS-1, visou cumprir os seguintes objectivos:

- Promover uma separação nítida entre os mecanismos de baixo nível, que dão acesso aos componentes executáveis da aplicação, através de dispositivos sensores e actuadores, e as funcionalidades de monitorização de mais alto nível, isto é, próximo do nível das abstracções das interfaces de utilização e das funcionalidades suportadas pelas ferramentas de análise e controlo das aplicações (por exemplo, para depuração de erros, perfilar o comportamento, ou controlar a configuração das aplicações dinamicamente).

- Servir de base para a definição de uma arquitectura virtual[1] distribuída, organizada segundo uma hierarquia de processos[2], de modo a disponibilizar as funcionalidades pretendidas, ao nível global da arquitectura distribuída.

[1] Isto é, realizável independentemente dos mecanismos específicos de cada plataforma computacional específica.

[2] Servidores.

- Promover a maior neutralidade possível, dos componentes da arquitectura de monitorização, relativamente aos cenários de utilização específicos. Isto é, em vez de incorporar na arquitectura, um conjunto de funcionalidades, definidas à partida, que se pensassem serem úteis, para a generalidade dos cenários de observação e controlo, preveu-se que a arquitectura que pudesse ser flexível para ser expandida com novas funcionalidades, quando necessário.

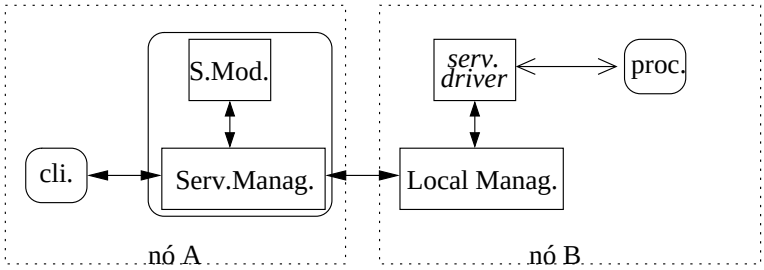

Figura 4.1: Arquitectura do primeiro modelo DAMS

Para suportar esta flexibilidade de expansão, definiu-se uma arquitectura a "duas dimensões" ou a "dois níveis", como ilustrado na fig. 4.1:

1. Um *nível básico* de suporte à distribuição, neutro e genérico, organizado em termos de um coordenador central, designado por *Service Manager* (SM), e por uma colecção de representantes, ditos *Local Managers* (LM), um local a cada nó da arquitectura, sendo que, o SM e os LM não suportam quaisquer serviços específicos de monitorização. As únicas funcionalidades do SM são a interacção com as ferramentas, e o encaminhamento dos pedidos destas para os respectivos LM, nos devidos nós, bem como a propagação das correspondentes respostas. As responsabilidades dos LM são as de servirem de representantes do SM, em cada nó, servindo de intermediários entre aquele e os sensores/actuadores, que interagem com os processos da aplicação locais ao nó. Para além disso, cada LM disponibiliza funções sobre o estado local dos recursos do seu nó, como sejam, acções para activação de processos, sua terminação e consulta dos processos existentes.

2. Sobreposto ao nível básico, a arquitectura DAMS-1 tem um *nível de suporte específico à monitorização*, organizado em unidades funcionais, chamadas "serviços". Cada um destes é realizado como um par (*Service Module* (SMod), *Service Driver* (SD),

responsável por encapsular a funcionalidade de um determinado serviço específico de monitorização. O SMod define uma interface API, que o SM torna disponível do lado dos clientes da DAMS-1 (interfaces de utilização e ferramentas). O SD, sob o controlo do LM local, é o responsável pelas acções de baixo nível correspondentes às funções da API do serviço, tais como, coligir informação, se for do tipo sensor, ou aplicar comandos, se for do tipo actuador, a serem aplicadas aos processos da aplicação alvo, por pedido do SMod.

A flexibilidade deste modelo resulta do facto de as funcionalidades específicas de monitorização só serem conhecidas dos pares (SMod,SD). A sua extensibilidade resulta da facilidade com que se podem adicionar novas funcionalidades a uma configuração, por simples inclusão de novos pares (SMod, SD) e pelo seu registo junto do SM. Todas as interacções entre um SMod e o(s) seu(s) SD associados, fazem parte da especificação do serviço. Torna-se, assim, possível ter múltiplos serviços coexistentes numa mesma configuração DAMS-1, mas aplicando diferentes estratégias de controlo ou de observação, ou interpretando a informação relevante a distintos níveis de abstracção.

Um primeiro protótipo, do modelo DAMS-1, foi realizado no contexto de um trabalho de alunos finalistas de Engenharia Informática [99], que o implementaram sobre a plataforma PVM, numa rede local de computadores Linux e mais tarde também em OSF/1. Sobre essa realização, definiu-se uma infraestrutura para a observação e o controlo de processos distribuídos, com um primeiro protótipo limitado ao suporte de funcionalidades de depuração de programas, onde:

- qualquer processo alvo tem um identificador global;
- existe um conjunto de comandos para controlar e inspeccionar individualmente cada processo: *stop, continue, next, step, print, set, breakpoint*, etc.;
- existe um conjunto de comandos para gerir a configuração da infraestrutura e os processos sobre o seu controlo: juntar ou remover máquinas do sistema, ligar ou desligar o controlo sobre cada processo nessas máquinas.

Este sistema foi utilizado com sucesso para a implementação de várias ferramentas, e foi sendo estendido com outros serviços, mais específicos, para outras utilizações. Exemplos destas utilizações são discutidos no capítulo 6. Este trabalho abriu o caminho a novas

utilizações e permitiu obter experiência sobre a sua utilização, em particular nas vantagens identificadas:

- reutilização dos serviços, uma vez desenvolvidos;
- partilha desses serviços entre vários clientes;
- extensão do sistema com novas funcionalidades, pela adição de serviços.

Também permitiu identificar algumas limitações:

- esta arquitectura é demasiado rígida, ao possuir um processo servidor central (o SM) que suporta todos os SMod, e com o qual todos os clientes têm de interagir;
- tanto o SM como os LM possuem uma arquitectura monolítica fixa sem distinção ou separação entre a parte responsável pelas operações de gestão dos SMod e respectivos *drivers*, e as interacções via a plataforma que as suporta (este protótipo está intrinsecamente ligado à plataforma, o PVM no caso);
- os processos LM apenas servem de intermediários entre os Smod e os respectivos *drivers*, não podendo possuir mais competências, o que se revela uma limitação à extensibilidade deste modelo da DAMS; por outro lado, cada LM centraliza todos os pedidos e respostas envolvendo os *drivers* nesse nó;
- os diversos serviços seguem esta arquitectura (SMod—*driver*), não sendo possível outras organizações internas para os serviços.

Surgiu assim a ideia de um novo modelo que permitisse responder melhor a novas utilizações e novos requisitos, de uma forma mais geral e flexível, em particular relativamente à própria arquitectura interna do sistema. Este é o modelo que se apresenta a seguir nesta dissertação.

4.3 Objectivos e requisitos

Após a apresentação das várias dimensões e abordagens de monitorização de aplicações distribuídas (capítulo 3) e, no seguimento do primeiro protótipo antes apresentado, identifica-

se a necessidade da existência de uma infraestrutura que suporte as mais variadas ferramentas para observação e controlo das aplicações. Neste trabalho propõe-se uma, denominada DAMS[3] (*Distributed Applications Monitoring System*).

O modelo DAMS define os conceitos e uma arquitectura abstracta que permita concretizar numa infraestrutura em implementações específicas, o seguinte conjunto de funcionalidades:

- o suporte de monitorização da execução de aplicações paralelas e distribuídas, seja para observação (medição de parâmetros por amostragens; detecção de eventos; etc.), seja para fins de avaliação do desempenho, seja para suporte à depuração dos programas;
- o suporte do controlo da execução (controlo do fluxo e modificação do estado), por exemplo para a depuração ou para a adaptação dinâmica em execução[4] do comportamento das aplicações;
- facilitar o transporte das ferramentas para diferentes ambientes *software*/*hardware* oferecendo abstracções, se possíveis independentes da plataforma, que permitam suportar ferramentas, já existentes, em diferentes plataformas;
- a fácil extensão das funcionalidades suportadas pelo sistema de monitorização, para permitir a adaptação flexível e evolução da infraestrutura;
- o suporte de múltiplas ferramentas clientes (no ambiente de desenvolvimento ou durante a execução) permitindo o seu uso em simultâneo e facilitar a sua integração pela introdução de mecanismos de cooperação e coordenação entre as ferramentas, sob a forma de serviços.

No desenvolvimento de aplicações paralelas e distribuídas, verifica-se uma grande diversidade de plataformas (*hardware* e *software*) assim como de modelos de programação, havendo uma grande variedade de cenários de utilização deste tipo de sistemas. A abordagem escolhida neste trabalho foi, por isso, a de definir um modelo e arquitectura minimalistas

[3] O nome deste sistema, apesar de se tratar de um novo modelo e arquitectura, foi herdado do primeiro sistema desenvolvido.

[4] Vulgarmente chamado, em inglês, de *steering*.

—um sistema de base— que sejam o mais neutros possível relativamente, quer aos modelos de programação, quer às plataformas de suporte à execução das aplicações, tal que:

- se permita uma interface única com a aplicação mas possibilitando o acesso concorrente das várias ferramentas;
- as funcionalidades das ferramentas se possam abstrair das implementações particulares a cada plataforma que as suporta;
- se ofereçam funcionalidades comuns às várias ferramentas, que estas possam reutilizar, mas se permita, ainda assim, a coexistência de funcionalidades específicas para casos particulares de certas plataformas ou ferramentas;
- se permita assim a partilha do estado da aplicação, assim como do próprio monitor, entre as várias ferramentas, ou permitindo a realização de serviços adicionais de coordenação, se garanta uma visão consistente entre as várias ferramentas.

As funcionalidades acrescentadas ao sistema de base podem resultar, quer das necessidades da própria monitorização, mas também da utilização que as ferramentas fazem do monitor. Por exemplo, no primeiro caso podemos ter novos mecanismos para monitorização como a inclusão de funcionalidades para a instrumentação dinâmica; no segundo, a inclusão de funcionalidades para facilitar a interacção e a cooperação entre ferramentas.

Diferentes implementações destas funcionalidades podem existir, de forma transparente, para diferentes plataformas, facilitando o transporte das ferramentas para as diferentes arquitecturas/plataformas (de particular importância para oferecer as funcionalidades esperadas pelas ferramentas nos vários ambientes). No entanto, numa primeira abordagem, assumem-se algumas características da plataforma sobre a qual se trabalha, de forma a limitar as abordagens possíveis e assim poder definir uma base sobre a qual o modelo é definido e avaliado. Considera-se por isso que a aplicação é executada sobre uma arquitectura física distribuída, isto é, com vários nós (ou máquinas físicas) permanentemente ligados por um meio de comunicação, sendo que os protocolos da plataforma subjacente, sobre a qual se realiza cada implementação, nos garantem:

1. a detecção de falhas nas trocas de informação e respectivo reenvio quando necessário;

2. existe um caminho virtual via esse meio de comunicação, entre quaisquer nós (logo entre quaisquer dois processos), mesmo que envolvendo intermediários na sua realização;

3. os mecanismos de autenticação permitem ao utilizador da aplicação e do sistema de monitorização a interacção com qualquer nó onde se executem componentes da aplicação.

Situações de falhas na ligação entre os nós ou dos próprios nós não são consideradas, sendo tratadas pela plataforma em que a DAMS se baseie, podendo levar a situações em que as interacções entre as várias entidades da DAMS falhem, sendo reportadas como erros nas versões dos protótipos realizados.

Cada nó da arquitectura física atrás referida, pode incluir mais de um processador, mas estes são geridos pelo respectivo sistema de operação de forma transparente para a aplicação, assim como para o sistema de monitorização. Nas arquitecturas com múltiplos processadores que executam várias instâncias do sistema de operação (como por exemplo os *clusters*), estes são vistos como sendo constituídos por diferentes nós. Assume-se também que os vários nós utilizam sistemas de operação e plataformas *software* equivalentes, no sentido em que oferecem um conjunto de abstracções com uma mesma semântica (tipicamente serão sistemas do tipo do UNIX/POSIX [107]).

O caso de sistemas de nós heterogéneos, com diferentes características arquitecturais (32 bits vs. 64 bits, *byte order*, etc.), não serão considerados ao nível da arquitectura da DAMS. Estes aspectos serão considerados e tratados pela plataforma de suporte à comunicação, no caso das interacções entre os vários componentes distribuídos da DAMS; e tratados pelas ferramentas ou sistemas específicos, no caso do tratamento da informação recolhida, funcionando a arquitectura DAMS de forma neutra relativamente à informação que transporta.

4.4 Dimensões e características do modelo

Neste trabalho passamos a referir, como “sistema de monitorização” um qualquer sistema existente na arquitectura distribuída, que permita interagir, mesmo indirectamente, com a aplicação ou sua plataforma de suporte, quer para observar o estado de recursos ou características destes, quer também para permitir a sua alteração, com vista a modificar parâmetros,

variáveis ou a controlar a execução da aplicação (fig. 4.2).

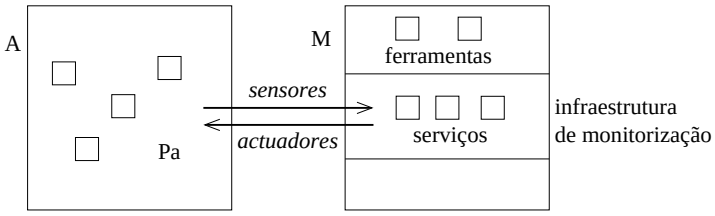

Figura 4.2: Relação da DAMS com a aplicação

Seguindo os princípios apresentados, o modelo proposto para este tipo de infraestrutura baseia-se numa arquitectura como a apresentada na secção 3.3.3, onde se tem uma separação entre sensores/actuadores, monitor e clientes (fig. 4.3). Procura-se a separação entre os vários componentes funcionais, de forma a permitir a separação das abstracções que oferecem, relativamente às respectivas implementações.

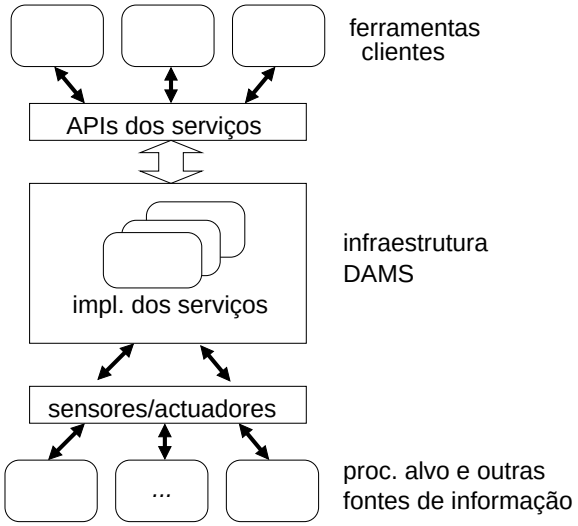

Figura 4.3: Modelo DAMS

A infraestrutura de monitorização da DAMS consiste num núcleo e numa colecção de entidades. À partida, consideram-se sempre presentes em qualquer sistema para observação ou controlo, baseado no modelo DAMS, os seguintes tipos de entidades:

ferramentas — clientes do sistema de monitorização, podendo haver vários clientes a aceder concorrentemente ao estado da aplicação alvo, quer para observação, quer para controlo. Esta entidade pode não ser uma ferramenta para o utilizador final, mas o

termo é aqui usado para referir qualquer entidade com necessidades de observação ou controlo da aplicação e que, para isso, recorre à infraestrutura DAMS.

sensores e actuadores — "dispositivos" (facilidades ou *drivers*) capazes de interactuarem directamente com a aplicação alvo e sua plataforma de suporte (*hardware* e *software*). Oferecem um conjunto de funcionalidades aos outros componentes do sistema de monitorização. Estes dispositivos podem consistir de funções internas aos sistemas de operação ou a funcionalidades adicionadas ao nível da aplicação ou das bibliotecas da plataforma usada (por exemplo por instrumentação dessas bibliotecas). Outros dispositivos podem tirar partido do próprio *hardware* como, por exemplo, os contadores ou outra informação existente no próprio *hardware* de alguns processadores.

serviços — estes referem-se a unidades agrupando funcionalidades, com estado interno, que estão disponíveis no sistema de monitorização. Funcionam como módulos na estruturação interna do sistema de monitorização e para permitir a sua extensão (ou simplificação) de uma forma estruturada. As funcionalidades oferecidas por cada serviço são acessíveis através da sua interface de programação pré-definida. Estes serviços permitem:

- definir um grupo de funcionalidades, num espaço de identificadores próprio, para uso de qualquer outra entidade;
- encapsular os detalhes arquitecturais e de implementação dessas funcionalidades sob a sua API;
- gerir recursos e permitir, se necessário, a sua partilha entre vários clientes;
- reduzir as dependências relativamente à plataforma de suporte, com vista a um mais fácil transporte para diferentes sistemas.

Estas entidades recorrem ao seguinte conjunto de funcionalidades, suportadas pelo núcleo da DAMS:

- a gestão e localização dos serviços na arquitectura distribuída;
- uma infraestrutura de comunicação que permita as interacções necessárias entre as entidades;
- a interacção com a plataforma de suporte para gestão das máquinas e processos sob observação.

O núcleo referido, tendo em conta os requisitos antes enunciados (secção 4.3), pretende-se mínimo. Com esta abordagem permite-se que o sistema possa ser estendido pela definição de novos serviços, os quais podem tirem partido das abstracções oferecidas pelo núcleo e por outros serviços. As próprias ferramentas baseiam-se nas funcionalidades oferecidas pelos serviços, sem necessidade de interacção directa com a plataforma ou acesso directo à aplicação alvo. Por seu lado os próprios serviços se podem basear noutros, formando vários níveis de funcionalidades, como se vê na figura 4.4.

4.4.1 Arquitectura abstracta

Destinando-se o modelo DAMS à monitorização de aplicações paralelas e distribuídas, as próprias entidades da DAMS encontram-se distribuídas, acompanhando os vários componentes da aplicação "alvo". Estas entidades acedem aos sensores e actuadores que interagem com os processos, aos sistemas de operação ou a outros recursos utilizados pela aplicação.

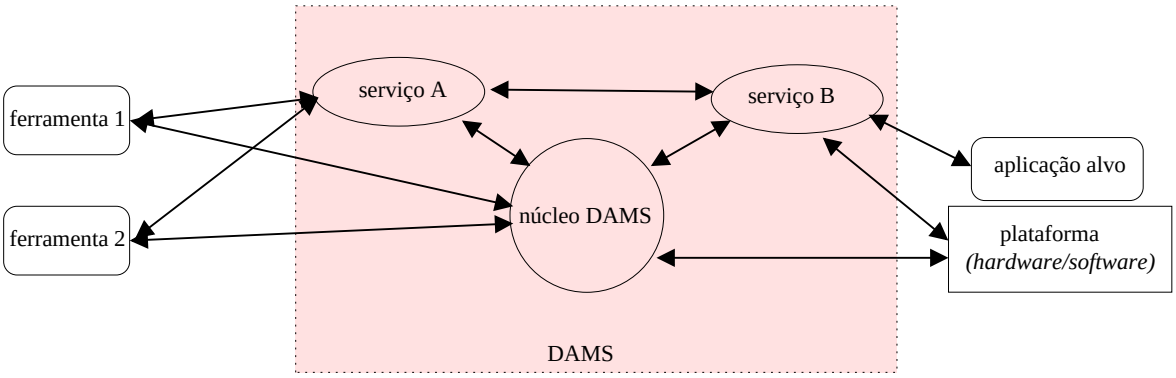

Figura 4.4: Modelo de arquitectura DAMS

O núcleo da DAMS tem ele próprio um representante em cada nó da arquitectura distribuída, que permite suportar a implementação das funcionalidades da infraestrutura DAMS, permitindo-se assim a interacção entre as entidades residentes nessa máquina e as restantes, serviços ou ferramentas clientes. O núcleo também suporta a gestão do conjunto de máquinas capazes de serem monitorizadas, definindo a máquina virtual da DAMS. Prevendo que este conjunto possa variar na sua composição, revela-se necessário que o núcleo ofereça primitivas para acrescentar ou retirar máquinas deste conjunto.

É assim que qualquer ferramenta cliente ou um qualquer serviço, pode obter a informação

sobre a máquina, ou os serviços disponibilizados em qualquer nó. As interacções entre as várias entidades, são suportadas usando o mecanismo de comunicação disponibilizado pelo núcleo. As funcionalidades oferecidas pelo núcleo são realizadas sobre a plataforma computacional de base, que suporta a execução das entidades presentes em cada nó.

4.5 Serviços

Um serviço é uma abstracção que representa, no modelo DAMS, um conjunto de funcionalidades relacionadas entre si, identificadas segundo um nome global único. Este nome é utilizado por todos os seus clientes para requerer o acesso ao serviço, ou seja, todos os que necessitam dessas funcionalidades devem conhecer ou poder obter o nome do serviço. Cada instância do serviço possui um estado interno, passível de ser partilhado por todos os seus clientes, quando se pretenda atender vários clientes em concorrência.

Os serviços permitem responder às necessidades de adaptação flexível a cada objectivo, através da definição, por configuração, do conjunto de serviços que devem estar presentes durante a execução da monitorização. Pode-se dispor de sistemas de monitorização com funcionalidades distintas, consoante os requisitos das ferramentas que se pretenda utilizar.

Os vários serviços podem desempenhar diferentes papéis no sistema DAMS, dependendo dos seus objectivos:

- locais – servem para interacção com a aplicação e respectiva plataforma num nó específico, como seja desempenhar o papel de interface com os sensores ou os actuadores, permitindo a integração destes na infraestrutura DAMS;
- globais – oferecem visões e funcionalidades globais à máquina virtual, permitindo centralizar o acesso partilhado ao estado e a funcionalidades dispersas pelos vários nós, sem necessidade de interagir explicitamente com cada um dos nós (serve assim de interlocutor central com vários serviços locais);
- intermediários – oferecem funcionalidades abstractas, mais para uso de outros serviços, do que para o uso pelas ferramentas. Podem por exemplo, agrupar vários sensores e actuadores num nó para oferecerem um novo tipo de monitorização nesse nó, para uso por outros serviços.

Um serviço que ofereça abstracções válidas em diferentes ambientes de computação paralela, ao ser disponibilizado nesses vários ambientes, permite que ao nível das ferramentas, não sejam visíveis as particularidades das plataformas. Estas ferramentas, ao dependerem apenas das interfaces com estes serviços, podem ser utilizadas em qualquer dos ambientes *hardware*/*software*, desde que os respectivos serviços estejam aí disponíveis. Os detalhes da arquitectura interna de cada serviço, tal como os da plataforma e dos sensores/actuadores, em cada caso, podem ser escondidos das ferramentas.

Como complemento às interacções com os serviços por pedidos de operações, e tendo em conta o carácter assíncrono dos sistemas paralelos e distribuídos, deve ser possível uma funcionalidade de interacção por meio de notificações assíncronas, que permita que os seus clientes, interessados, sejam informados de alterações no estado da aplicação ou da respectiva plataforma. Pode servir também para notificar a ocorrência de alterações no estado interno do serviço, eventos detectados pelo serviço, ou ainda possibilitar padrões assíncronos de interacção com o serviço. Para este efeito, o modelo DAMS disponibiliza um mecanismo de notificação assíncrona baseado em canais de eventos, apresentado na secção 4.6.3.

Cada serviço é realizado como um módulo funcional acessível a vários clientes. Na sua implementação pode permitir que estes usem concorrentemente essas funcionalidades, competindo ao próprio serviço a gestão dos acessos ao seu estado interno e aos recursos utilizados, assim como à própria aplicação alvo. Para tal, compreenderá um servidor que gere essa concorrência e as partilhas possíveis, com os seus vários clientes, numa arquitectura cliente/servidor. Na figura 4.5 é apresentada uma representação esquemática de um serviço.

A interacção dos clientes com cada serviço é suportada por uma interface de programação específica pré-definida (vulgarmente referida como API[5]), podendo esta esconder parte dos detalhes da interacção com o nível básico da DAMS e com os servidores que implementam parte do serviço. A utilização das funções oferecidas na implementação do serviço por parte de um cliente, exige que este requeira inicialmente à DAMS, a localização/instanciação do serviço, pela indicação do respectivo nome, sendo-lhe indicado pela DAMS o identificador do respectivo servidor a usar nas interacções com esse serviço, independentemente da localização do cliente ou do servidor que realiza esse serviço. A realização da API pode incluir operações locais ao próprio cliente, pode recorrer à infraestrutura da DAMS, assim como às funcionalidades do respectivo servidor que realiza o serviço. Possibilita-se a implementa-

[5]Do inglês *Application Programming Interface*.

ção sob essa API, de forma transparente para os seus clientes, de serviços com arquitecturas internas mais complexas.

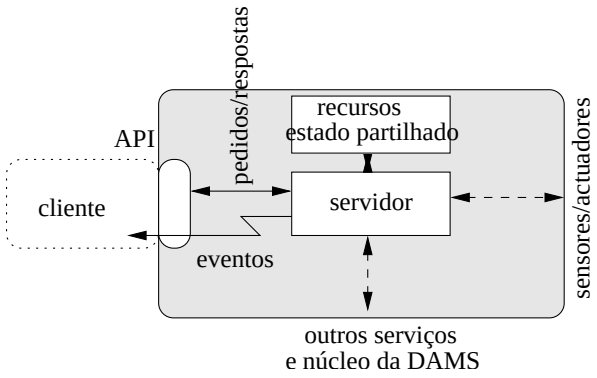

Figura 4.5: Esquema de um serviço

De seguida apresentam-se, em mais detalhe, as diversas interacções de um serviço com as restantes entidades no sistema DAMS.

4.5.1 Interacção com os sensores/actuadores

Os sensores e os actuadores em cada nó são responsáveis pela interacção directa com a aplicação e respectiva plataforma de suporte à execução (bibliotecas, sistema de operação ou mesmo o *hardware*). Esta interacção é possível recorrendo às capacidades específicas locais a esse nó, nomeadamente o sistema de operação, a instrumentação presente na plataforma, etc. Para que estes instrumentos possam ser utilizados pelas várias entidades na DAMS, independentemente da sua localização, necessitam de ser integrados no sistema. Para tal devem-se apresentar como um serviço e serem registados junto do núcleo do sistema DAMS no nó onde se encontram, a quando da sua inicialização.

Esta integração poderá ser conseguida por um serviço que serve de intermediário, acedendo aos sensores/actuadores através da interface própria por estes disponibilizada (fig. 4.6), ficando esses detalhes assim “escondidos” dos restantes serviços e das ferramentas.

Outra forma de integração poderá passar pela reescrita desses sensores/actuadores como verdadeiros serviços da DAMS (fig. 4.7), integrando-se plenamente na infraestrutura.

Noutros casos, bastará que esses sensores ou actuadores se tornem clientes da DAMS e

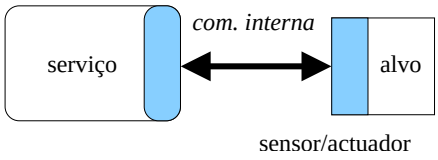

Figura 4.6: Um serviço como intermediário dos sensores ou actuadores

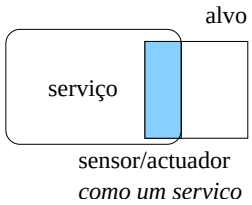

Figura 4.7: Sensor ou actuador integrando-se como um serviço

utilizem a interface de um serviço existente (fig. 4.8), para disponibilizarem informação ou funcionalidades ao restante sistema.

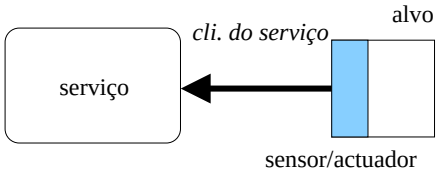

Figura 4.8: Sensor ou actuador cliente de um serviço

Torna-se possível que vários clientes partilhem estes dispositivos, pela utilização destes serviços, sendo que a sua implementação definirá as políticas necessárias de acesso entre os seus múltiplos clientes. Por exemplo, a interacção com os dispositivos de instrumentação da aplicação alvo é efectuada por intermédio de serviços implementados por servidores localizados em cada uma das máquinas e que acedem à aplicação e/ou plataforma que a suporta (tipo *device-drivers*). Cabe à implementação do serviço resolver, de acordo com cada caso, os problemas associados aos acessos concorrentes de vários clientes.

4.5.2 Interacção com os clientes

Os clientes de cada serviço (ferramentas ou outros serviços) vão recorrer a uma infra-estrutura oferecida pela DAMS para interactuar com estes. Por facilidade de operação, tal deve ser normalmente escondido sob a API do serviço. Tal como se viu na secção 4.4, as ferramentas devem interagir com o sistema de monitorização por meio das interfaces de cada serviço, facilitando o seu transporte. A API de cada serviço esconde a sua arquitectura interna e os detalhes da interacção com o respectivo servidor, utilizando a DAMS. Por exemplo, determinado serviço ao ser requisitado pode, na implementação da sua API, definir qual o servidor que irá realmente atender os futuros pedidos vindos deste cliente, possibilitando distribuição da carga do serviço. Pode também incluir de forma transparente, operações de coordenação com outras ferramentas que usem o mesmo serviço.

Este tipo de interface cobre os requisitos das interacções que seguem o padrão de pedido/resposta, identificando-se dois casos:

- por execução de operações implementadas na API no contexto do próprio cliente, sem qualquer interacção com outras entidades;
- por pedidos efectuados ao servidor que realiza o serviço, que corresponde à execução de procedimentos remotos.

Para esse último caso há necessidade de, através do núcleo DAMS, o cliente poder localizar o servidor associado ao serviço pretendido. Logo, também o serviço tem de, através desse mesmo núcleo, registar a disponibilidade do respectivo servidor (ou servidores, dependendo da sua arquitectura interna), para que possam ser localizados pelos clientes.

Nas interacções baseadas em notificações assíncronas, deve o cliente poder indicar o seu interesse nessas notificações e registar a sua operação de recepção (e possível tratamento), dessas notificações. Para tal, as entidades envolvidas recorrem a canais de eventos, sendo estes implementados e geridos pelo núcleo da DAMS, desempenhando estes canais o papel de intermediários entre os fornecedores de eventos e os possíveis interessados, permitindo tornar a sua interacção assíncrona. Tipicamente o serviço cria o canal de eventos junto do núcleo e disponibiliza este aos seus clientes. Estes podem subscrevê-lo, indicando junto do núcleo, a sua operação de recepção.

Um evento gerado por um serviço pode servir para:

- notificar alterações no estado da aplicação ou sua plataforma sob a monitorização do serviço;
- notificar alterações no estado do próprio serviço, consequência da situação anterior ou da interacção com outros clientes;
- notificar a disponibilidade de informação em consequência de pedidos prévios dos cliente. Este caso permite que, caso a caso, se possam implementar pedidos em que o cliente não se bloqueia, sendo a resposta entregue mais tarde por via de uma notificação.

4.5.3 Interacção com o núcleo e outros serviços

Cada serviço, na sua implementação, pode recorrer a qualquer outro serviço disponível. Toma assim o papel de cliente desse serviço, recorrendo para tal à API por esse disponibilizado, tal como visto na secção anterior.

De forma idêntica, para requerer as operações providenciadas pelo núcleo da DAMS, recorre à respectiva API. Note-se que o núcleo providencia as operações necessárias para a gestão de recursos, permitindo abstrair, ao nível do serviço, os detalhes da plataforma usada. Só será necessário recorrer a essa plataforma, ou aos sensores/actuadores, naquilo que será o papel específico do próprio serviço enquanto membro do sistema DAMS, na implementação das funcionalidades por este disponibilizadas às restantes entidades do sistema.

4.6 O núcleo da arquitectura DAMS

É da responsabilidade do núcleo da DAMS, como visto nas secções anteriores, o suporte das funcionalidades básicas que permitam a implementação de um sistema DAMS, incluindo a gestão e identificação de todas as entidades. A sua implementação compreende, tal como no caso dos serviços, uma parte servidora e uma interface cliente, disponível a qualquer entidade que necessite de requerer as funcionalidades do núcleo (ver figura 4.9).

Para cada cenário de utilização, existirá uma configuração adaptada ao uso pretendido. Esta consistirá num conjunto de máquinas sob observação, podendo este variar no tempo; e um conjunto de serviços disponíveis em cada máquina e acessíveis a quaisquer ferramentas.

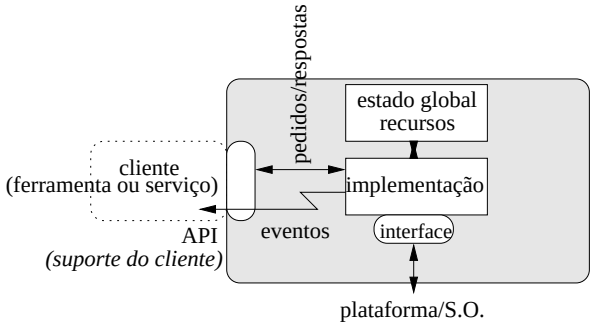

Figura 4.9: Esquema do núcleo

Nesta secção apresenta-se a arquitectura deste núcleo e o conjunto de funcionalidades mínimas oferecidas. O núcleo terá de estar acessível em todas as máquinas sob monitorização e de providenciar o suporte necessário para disponibilizar os serviços que se pretendam oferecer via DAMS. Este suporte necessita então de contemplar:

- uma infraestrutura básica de comunicação para suportar a interacção entre entidades, nomeadamente o envio de pedidos aos serviços, cuja implementação se pode encontrar em outro nó da arquitectura distribuída, bem como o envio da resposta ou outras trocas de informação.
- a gestão de recursos e entidades:
  - suportar o registo dos serviços, a localização da sua implementação e a ligação por parte dos respectivos clientes (possivelmente remotos); é assim necessário identificar cada entidade de forma unívoca;
  - uma noção de canal de eventos; esta forma de interacção permite, usando um modelo de subscrição, a notificação assíncrona de eventos por todas as entidades interessadas;
  - gestão da máquina virtual que compreende as máquinas sob monitorização, onde é possível observar ou controlar componentes da aplicação alvo e sua plataforma de suporte. Cada máquina a monitorizar terá de oferecer a infraestrutura básica descrita, por forma a permitir a interacção das entidades locais com as restantes presentes no sistema DAMS.

A infraestrutura da DAMS que permite as interacções entre todas as entidades assenta

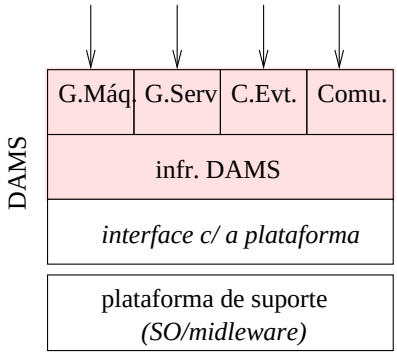

Figura 4.10: Núcleo da DAMS

numa interface adaptada à plataforma onde se executa (ver fig. 4.10), procurando assim manter-se a independência entre o núcleo da DAMS, seus serviços e clientes, em relação à plataforma concreta sobre a qual determinada implementação da DAMS se executa. Vejamos, de seguida, as várias funcionalidades providenciadas pelo núcleo da DAMS aos serviços e seus clientes.

4.6.1 Gestão de serviços

Na realidade cada serviço pode ser suportado por um servidor ou por um conjunto de servidores, dependendo da sua arquitectura interna, sendo da responsabilidade do próprio serviço a sua implementação. Este deve manter esses detalhes escondidos dos clientes sob a respectiva API de acesso. O identificador da componente servidora registada junto do núcleo, será o da entidade de interface, sendo depois responsabilidade da implementação desse serviço o modo de gerir a sua arquitectura interna e como implementar a API oferecida ao cliente. O cliente estará, tipicamente, a interactuar com um destes servidores quando requer qualquer operação do serviço, através da API que se encontra implementada na componente servidora do serviço.

Na implementação dos serviços, a sua componente servidora, para poder disponibilizar as suas funcionalidades à respectiva API nos seus clientes, recorre ao núcleo da DAMS para se dar a conhecer e poder ser referenciada por qualquer um desses clientes. O serviço tem de possuir um identificador global, independente da sua localização, e cada operação oferecida tem de ser identificada no serviço, permitindo a cada cliente designar o pedido pretendido. Para que as entidades clientes possam verificar a existência da componente servidora do

serviço, devem no início poder encontrá-la e contactá-la. Para tal, o núcleo DAMS mantém um registo dos serviços disponíveis, ou seja, cada servidor que realiza um serviço efectua o seu registo, indicando a esse núcleo o seu identificador e o nome pelo qual o serviço é conhecido. Este é o nome que é usado como chave pelos seus clientes quando pretendam encontrar o serviço, e verificar a sua disponibilidade.

A gestão destes servidores que suportam a implementação dos serviços compreende as seguintes operações, aqui descritas de forma abstracta:

srvId NewService(op-table) — dada a lista de funções `op-table`, que definem a interface de um servidor de serviço, cria esse novo serviço. Na realidade a realização desta operação será específica do nível da plataforma de base que suporta a DAMS, e garantirá que estas operações são devidamenta registadas a esse nível, tornando-as acessíveis às entidades remotas. O identificador do serviço, atribuído pela plataforma de base, é retornado num tipo opaco `srvId`.

RegService(servname, srvId) — regista um serviço na DAMS segundo o nome indicado, estando a respectiva componente servidora disponível com o identificador indicado, obtido da operação anterior.

srvId GetService(servname) — procura o serviço de nome indicado, devolvendo o respectivo identificador caso este se encontre registado. Este identificador é o registado na operação anterior.

FreeService(srvId) — indica à DAMS que o processo invocador já não pretende utilizar mais o serviço indicado (é assim possível à DAMS manter informação sobre a utilização dos serviços).

Por questões de desempenho e gestão dos recursos do sistema distribuído, deve procurar-se que o serviço disponha de um servidor na mesma máquina que o cliente. De notar também que a partilha de recursos entre a infraestrutura de monitorização e a aplicação pode ter efeitos de interferência. não desprezáveis sobre esta última, com consequências no desempenho e no comportamento lógico da execução observados. No entanto, por necessidades do próprio serviço ou localização do cliente, estes vão estar muitas vezes em máquinas diferentes. O tratamento destas situações é deixado ao cuidado dos serviços e das ferramentas, para que estes implementem as soluções adequadas à natureza de cada uma, de acordo com os objectivos pretendidos em cada caso.

4.6.2 Gestão da máquina virtual de monitorização

Para ser possível a interacção dos sensores e actuadores com a aplicação ou a sua plataforma de suporte, exige-se que estes estejam presentes em cada máquina monitorizável. Todas as máquinas onde se localizam estes elementos, passíveis de participarem no sistema DAMS, integram uma máquina virtual que fica sob controlo da DAMS. Como consequência, pelo menos parte desse núcleo (e o suporte da infraestrutura básica) deve estar representada em cada uma destas máquinas para suportar a implementação das noções antes apresentadas e permitir o acesso aos serviços locais.

A infraestrutura DAMS permite gerir as máquinas que fazem parte desta máquina virtual, acrescentando mais máquinas ou removendo as existentes, consoante os interesses/necessidade das ferramentas clientes. Para tal, serão dinamicamente lançados ou terminados os processos que suportam o núcleo DAMS em cada máquina.

Esta noção de máquina virtual é idêntica à existente em muitas plataformas que suportam as aplicações paralelas e distribuídas. Neste caso, uma representação local do sistema de operação, ou processos auxiliares e bibliotecas, dependendo do nível de abstracção e da arquitectura, implementam uma noção de máquina virtual para suporte da execução do programa distribuído.

O representante local do núcleo da DAMS tem acesso directo a um conjunto de informação e recursos da máquina e permite torná-los acessíveis ao resto do sistema, incluindo, por exemplo:

- o nome da máquina;
- a lista de processos na máquina;
- a lista de máquinas conhecidas;
- a lista de serviços disponibilizados localmente;
- qualquer outra informação sobre a máquina física como, por exemplo: arquitectura, número de processadores, etc.

As operações suportadas pelo núcleo para a gestão desta máquina virtual incluem:

hostId AddHost(hostname) — pedir a adição de um nó físico (máquina) ao sistema DAMS. Este processo exige que o núcleo DAMS recorra à interface com a plataforma ou do sistema de operação para aceder às facilidades que permitam a execução de uma nova instância do núcleo DAMS no nó cujo nome foi indicado. É devolvido o identificador (`hostId`) da nova instância DAMS nesse nó.

hostId FindHost(hostname) — obter a identificação do nó cujo nome é indicado. Esse nó tem de estar instanciado no sistema DAMS (operação anterior).

DelHost(hostId) — pedir para terminar (remover do sistema DAMS) o nó (normalmente remoto) cujo identificador é indicado.

hostIdList ListHosts( ) — pedir a lista de identificadores dos nós DAMS, conhecidos desta máquina.

procIdList ListProcs(hostId) pedir a lista de todos os processos locais ao nó.

hostInfo damsHostInfo(hostId) obter informação genérica sobre o nó.

Para a interacção da DAMS com a plataforma e o sistema de operação, existe uma camada de interface que abstrai dos respectivos detalhes (descrita na secção 4.6.4).

Uma instrumentação ao nível da plataforma que suporta o núcleo pode detectar as alterações nessa máquina virtual e difundir essas alterações sob a forma de eventos. Um serviço ou uma ferramenta cliente, se interessados, podem utilizar esta informação para reflectir estas alterações no seu próprio estado interno.

4.6.3 Canais de eventos

A notificação das ferramentas sobre a ocorrência de determinados eventos, correspondentes a alterações no estado da aplicação, ou do sistema de monitorização, exigem um tipo de interacção assíncrona. Estes eventos podem também corresponder a alterações no sistema DAMS ou nos seus serviços, ou até em outras ferramentas. Estes acontecimentos não são normalmente previsíveis e ocorrem em concorrência com o funcionamento normal das ferramentas e dos serviços, mas os interessados devem poder ser notificados desses acontecimentos.

Também as interacções serviço/cliente podem ter de recorrer à notificação de eventos, feita de forma assíncrona, como visto antes. Para tal, pode ser criado na DAMS um "canal de eventos", que uma entidade interessada pode subscrever, para passar a receber (assincronamente) as notificações geradas por quem as detectou. Este mecanismo é disponibilizado a cada entidade no sistema.

A DAMS suporta por isso uma noção elementar de evento, correspondendo a uma mensagem urgente contendo a informação a notificar. O sistema DAMS é responsável por implementar a noção de canal de eventos como uma entidade perante a qual um determinado tipo de eventos é reportado e em relação aos quais outras entidades podem declarar-se subscritoras.

A noção de canal permite cumprir dois objectivos:

- funciona como intermediários entre emissor e receptores, separando-os, evitando a necessidade de se sincronizarem explicitamente;
- divide o espaço dos eventos, utilizando-se canais diferentes consoante o tipo de notificações ou canais específicos para cada serviço.

Sendo o seu objectivo a difusão desta informação, os canais fazem a propagação de cada evento recebido a todos os seus subscritores, assim que este é entregue/criado, sem bloqueio do emissor. A entrega deve ser assíncrona, podendo recorrer na sua implementação a sinais (como no sistema UNIX) ou à invocação de rotinas registadas pelos interessados (*callback* de *handlers*) para a entrega das notificações, podendo ainda utilizar *threads* dedicados à sua recepção, conforme o suporte disponível na plataforma e no sistema de operação subjacentes.

Estes canais devem podem ser encadeados, tornando-se um canal subscritor de outro, permitindo-se o seu encadeamento para a propagação dos eventos. Esta utilização introduz flexibilidade relativamente à arquitectura da máquina virtual permitindo propagar os eventos entre máquinas enquanto se propagam entre canais em máquinas diferentes.

Este encadeamento permite também a utilização de canais intermédios que funcionam como concentradores ou como difusores destes eventos (fig. 4.11). Fica também aberta a possibilidade de utilização de serviços intermediários que filtrem ou processem estes eventos, antes de os propagarem para os respectivos subscritores finais. Este tipo de infraestruturas será iniciada, tipicamente, pelos próprios serviços.

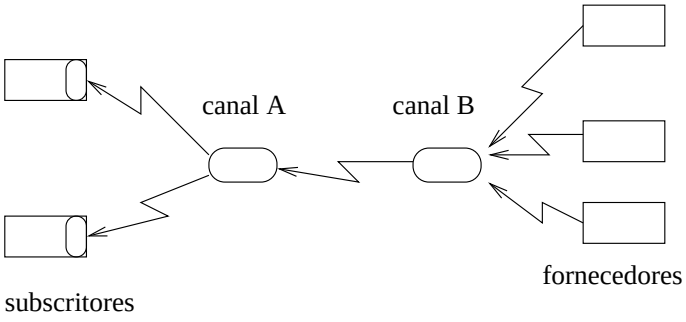

Figura 4.11: Rede de canais de eventos

O núcleo DAMS oferece um conjunto de operações, que permite a criação de canais de eventos, o registo de rotinas de tratamento (*handlers*) no contexto dos clientes e a notificação dos eventos. Estas operações fazem parte da API do núcleo, disponibilizada a todas as entidades.

A criação de novos canais usa uma chave como identificador do novo canal, ou permite o acesso a este, caso já tenha sido criado. Cada serviço pode requerer a criação de um canal e depois passar a respectiva identificação aos seus clientes, para que estes possam requerer a subscrição. As operações sobre canais de eventos são descritas a seguir de forma abstracta:

EvCh NewEvCh(key) — é obtido, do núcleo DAMS, o identificador do canal para notificação de eventos, que tenha a chave indicada. Se este não existe, um novo canal é criado com essa chave.

DelEvCh(EvCh) — é destruído o canal de eventos indicado. Deixa de ser possível notificar novos eventos via este canal.

PutEv(EvCh, event) — o evento descrito pelo argumento `event` é colocado no canal indicado (o evento pode ser qualquer tipo de dados, opaco para a DAMS). O canal de eventos deve *a posteriori* proceder ao seu envio para todos os subscritores.

Este modelo pressupõe que cada cliente pode invocar, através da interface fornecida pela DAMS, os mecanismos que lhe permitem registar as funções de tratamento (*handler*), e que, consoante as possibilidades da plataforma de suporte, permite a chamada dessas funções assíncronamente em relação à execução normal do cliente. Por exemplo, uma solução pode ser baseada num modelo de *threads*, ou oferecer-se na interface, uma operação para teste (*polling*) desses eventos:

SetHandler(EvCh, Handler) — o canal de eventos indicado é subscrito pelo invocador, pela associação do *handler* indicado ao canal.

DelHandler(EvCh, Handler) — o canal indicado deixa de estar subscrito via o *handler* indicado. Este é removido da subscrição do canal.

Handler EvCh2Handler(EvCh ech) — obter *handler* correspondente à operação que permite colocar um evento no canal. Permite que este canal possa ser subscritor de outro.

Handler Func2Handler(function) — dada a função indicada, é criado um *handler* capaz de ser usado na subscrição de um canal de eventos. Esta função ficará registada na interface DAMS como um possível *handler*, sendo-lhe atribuído um identificador que permita o seu uso nas operações anteriores.

4.6.4 Interface com a plataforma de suporte

Além do núcleo DAMS servir de suporte aos serviços, permitindo isolá-los da plataforma concreta sobre a qual se executam, necessita de servir de infraestrutura à interacção entre as várias entidades no sistema DAMS. Esta infraestrutura deve assim, permitir a identificação de todas as entidades e a sua interacção, independentemente da localização. Deve também, oferecer suporte para o acesso aos recursos oferecidos pelo sistema de operação, permitindo independência do próprio núcleo relativamenta à plataforma sobre a qual se executa. Vejamos os requisitos assumidos para estes dois aspectos.

Suporte às Interacções

Dois tipos de interacções entre as entidades são identificáveis:

1. pedidos de operações (com ou sem resposta);
2. notificações assíncronas (com registo da função de tratamento ou *handler*).

Para cada realização concreta da DAMS assume-se a existência, na plataforma de base, de primitivas capazes de trocar informação entre processos, mesmo que em máquinas remotas, permitindo a codificação dos pedidos e fazando-os chegar à entidade que os executa. De forma idêntica, esta plataforma tem que permitir a codificação das respostas e devolvê-las à

respectiva entidade que efectuou o pedido. Por outro lado, na chegada de uma notificação, deve ser capaz de desencadear a execução da respectiva função de tratamento, nas entidades subscritoras.

A identificação das entidades tem de contemplar as diversas situações, tendo em conta a possibilidade de mais de uma poder residir no mesmo processo, e que estes processos se encontram distribuídos por diversas máquinas. Assim, esta identificação tem de permitir distinguir a máquina, o processo e, finalmente, a entidade dentro desse processo.

As primitivas de comunicação da plataforma de base devem permitir implementar a interacção entre as várias entidades, incluindo, por exemplo, os pedidos cliente/servidor na implementação das API de cada serviço. Isto pode ser conseguido, com base num modelo de chamada de procedimentos remotos (como os RPC[6]), ou num modelo baseado em mensagens. Em qualquer dos casos a camada interna do núcleo da DAMS, encarrega-se de fazer a correspondência entre operações que suportam as interacções ao nível da DAMS e as primitivas do modelo de comunicação subjacente.

No caso das notificações, uma entidade, o canal de eventos, serve de intermediária entre quem envia e quem recebe as notificações. Esta entidade possui, na sua componente servidora, a implementação necessária para que a operação de envio de cada evento, a usar pelo gerador do evento, desencadeie a sua transmissão aos respectivos subscritores. Esta operação não devolve valores de retorno para o cliente gerador do evento, permitindo tornar a entrega da notificação aos destinatários assíncrona, relativamente ao seu envio pelo seu gerador.

A recepção em cada subscritor é desencadeada pelo canal de eventos que chama a respectiva função registada pelo subscritor. Este mecanismo é idêntico ao indicado para o despacho de pedidos, sendo que: a quando da subscrição do canal de eventos, e no lado do subscritor, ficou registada uma função gerida pelo núcleo DAMS, que pode ser invocada pelo canal de eventos; o seu identificador é o passado ao canal como *handler* de cada evento.

O receptor efectua previamente a subscrição do canal, sendo nessa operação realizadas as seguintes acções:

1. a função de recepção das notificações é registada na interface do cliente para o canal indicado, significando que a operação que recebe os eventos pode identificar qual a

[6]Remote Procedure Call [118]

função a chamar quando for recebida uma notificação com origem nesse canal;

2. o identificador da operação que recebe as notificações é registado, junto do canal, na lista de subcritores.

No caso das ferramentas, sendo entidades apenas clientes do sistema, podem na sua implementação não incluir um ciclo de recepção e atendimento, dado que não atendem pedidos vindos das restantes entidades. Caso seja necessário atender pedidos de outras entidades, estamos perante uma oferta de operações às restantes entidades e devemos, por isso, transformar esta ferramenta num componente servidor de um novo serviço. No entanto pode-se desenhar um mecanismo para desencadear a recepção da notificação de forma idêntica à usada nos servidores. Tal pode ser conseguido por uma primitiva de teste explícito da disponibilidade de novos eventos que inclui o despacho desses eventos. Esta primitiva pode ser invocada por um *thread* interno à ferramenta e que fica permanentemente atento à chegada de notificações (qual ciclo de servidor), ou por uma chamada explícita pelo programa na implementação da ferramenta.

Acesso ao sistema de operação

O núcleo DAMS, para suportar a implementação das operações oferecidas aos serviços e sua infraestrutura, necessita de ter acesso a alguns recursos oferecidos tradicionalmente pelos sistemas de operação (ou outra plataforma). Em particular, na consulta de informação sobre as máquinas e processos que executam, assim como para a sua gestão. Para tal assume-se um conjunto mínimo de funcionalidades numa camada abstracta interna, sobre a qual se implementa o núcleo e que permite a consulta, criação e destruição de processos, e o lançamento de processos em máquinas remotas, como sejam:

```
Exec(prog), Kill(pid), RemoteExec(host,prog), ListProcs(), ProcInfo(),
HostInfo()
```

Esta camada é chamada, no caso do corrente protótipo, de `dams-low`, sendo apresentada na secção 5.5.2.

Conclusão

A implementação de cada serviço deve tomar por base uma infraestrutura, idealmente comum em qualquer ambiente onde o sistema DAMS execute. No entanto, dadas as diferenças entre as plataformas e as funcionalidades que podem à partida ser oferecidas em cada caso, torna-se difícil tomar uma já existente como o mecanismo base de interacção para toda a DAMS. Pode-se definir de raiz um suporte próprio, implementado em todos os casos, ou optar por adaptar a DAMS e seus serviços ao suporte disponível em cada caso. Os princípios apresentados nesta secção servem para definir o conjunto de funcionalidades que os serviços podem esperar existir e servem para limitar os tipos de interacções possíveis para reduzir os requisitos da infraestrutura de comunicação. Pode-se assim, e dependendo de cada caso, optar por implementar como parte da DAMS as primitivas para troca de mensagens apresentadas sobre as primitivas do sistema de operação ou outra plataforma disponível; ou pode-se fazer corresponder logo as operações exportadas por cada serviço numa infraestrutura tipo RPC existente, utilizando a DAMS uma camada de adaptação entre o servidor do serviço, suas operações e essa mesma infraestrutura.

Esta problemática é mais discutida no capítulo seguinte, onde se abordam as possibilidades de implementação sobre uma plataforma de troca de mensagens ou sobre mecanismos do tipo RPC.

4.7 Exemplos de serviços

As funcionalidades antes apresentadas servem de base à construção de sistemas de monitorização, mas apenas nas suas dimensões de gestão de entidades e de máquinas, e de interacção, incluindo formas síncronas e assíncronas. Mas tendo estas funcionalidades um carácter genérico, coloca-se a questão de inclusão ou não, dentro deste núcleo DAMS, das funcionalidades adicionais específicas do suporte à monitorização. A opção tomada, em consonância com a ideia minimalista e de flexibilidade que se procura seguir, foi a de não incluir estas funcionalidades no núcleo, mas sim, elas próprias, como serviços.

Nesta secção apresentam-se dois serviços, representativos das duas principais classes de funcionalidades requeridas por um sistema destes: um de observação e outro de controlo. Estes podem ser considerados como a base para as mais variadas utilizações. Estes revelam-

se bastante gerais e passíveis das mais variadas utilizações, as quais são aqui introduzidas e depois discutidas no capítulo 6.

Assegura-se assim a maior flexibilidade possível. Estes serviços podem ser usados na construção de ambientes de monitorização ou controlo mais elaborados, juntando ao sistema novos serviços que usam estes. No entanto, se necessário, estes próprios serviços mínimos podem ser removidos do sistema, ou substituidos por outros, visto o núcleo da DAMS não estar comprometido com a sua existência.

4.7.1 Serviço de traços

Uma das técnicas mais usada para observação de aplicações é sem dúvida a recolha de um traço da sua execução. Muitas técnicas têm sido usadas: desde a simples introdução de "`printf`"s ou funções equivalentes no código fonte, até sistemas autónomos que a nível do sistema de operação ou por alteração da aplicação, recolhem e armazenam durante a execução, todo o tipo de eventos que permitam descrever as mais variadas transições de estado de todo o sistema.

Estas técnicas são usadas para, por exemplo (visto no capítulo 2):

- visualizar e ajudar à compreensão do comportamento da aplicação;
- ajudar na identificação e localização de erros, assim como de quais as suas possíveis causas;
- efectuar medições e contagens de valores que permitam obter o perfil da execução da aplicação com vista a melhorar o seu desempenho; ou identificar problemas de concepção ou implementação no seu desempenho ou no uso dos recursos;
- permitir suportar serviços de adaptação dinâmica da aplicação à plataforma computacional (balanço de carga, tolerância a falhas, etc.);

Normalmente os sistemas de observação necessitam de um mecanismo para coligir e registar uma sequência de eventos que permitam descrever a execução da aplicação — o seu traço de execução. Este mecanismo permite coligir e canalizar para as ferramentas, uma sequência de registos que descrevem esse traço, possivelmente com armazenamento temporário dos mesmos.

Seguindo as ideias antes apresentadas, concebeu-se um serviço para o encaminhamento de traços de execução (ou qualquer outra sequência de registos). A ideia base para este serviço é a de dispor de um simples canal que recebe registos e, posteriormente, os difunde para os seus consumidores. Este serviço pode servir de base para a implementação de funções de monitorização mais complexas.

Este serviço necessita de ser neutro em relação ao conteúdo dos registos que transporta, para poder ser usado nas mais variadas situações. Terá de ser o consumidor final da informação a interpretá-la. O serviço assegura apenas o fluir dos registos da sua origem para os seus consumidores. Assim se garante a transparência deste serviço e se torna possível o seu uso para qualquer tipo de registos e traços de execução.

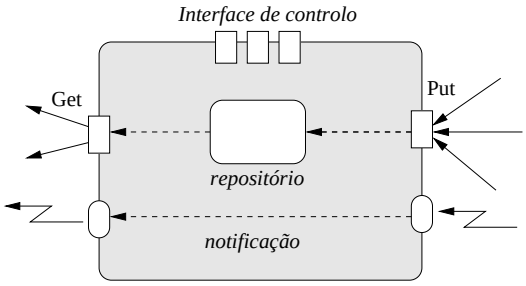

Figura 4.12: Um canal para traços

Este trata-se de um serviço local a cada máquina, possuindo um repositório, parametrizável quanto ao seu tamanho, para reter temporariamente os registos, até os passar aos consumidores.

Estes serviços podem ser encadeados formando uma árvore que encaminha os registos recolhidos por todo o sistema, até à ferramenta que os pretende obter. Estas conexões permitem desenhar topologias que melhor se adaptam à arquitectura do sistema monitorizado e às necessidades das ferramentas. Os sensores responsáveis pela recolha de informação (a instrumentação p.e.) em cada máquina, devem ser capazes de entregar essa informação no serviço de traço nessa máquina. Por sua vez este fará, de acordo com parâmetros de funcionamento, fluir esta informação para o respectivo canal consumidor, e assim sucessivamente até chegar à raiz onde a ferramenta recebe essa informação.

Um monitor do tipo já descrito no capítulo 3, pode ser construído usando várias instâncias destes serviços segundo a arquitectura idêntica à apresentada na figura 4.13.

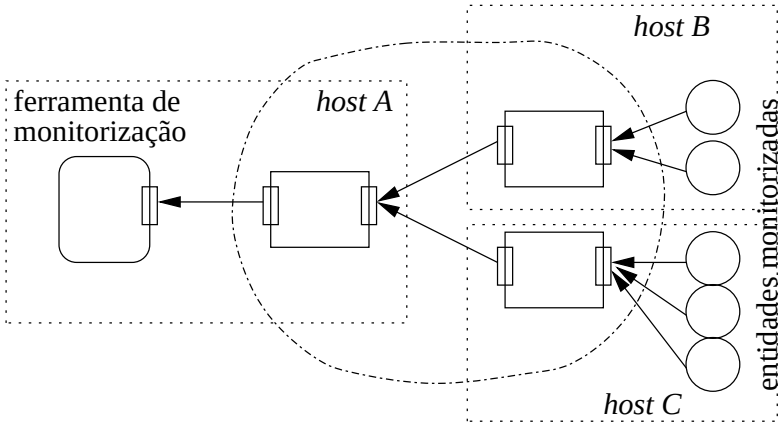

Figura 4.13: Monitorização usando uma rede de serviços de traço

Caso a ferramenta só pretenda o traço da execução completa da aplicação, para análise *post-mortem*, só vale a pena enviar para a ferramenta a informação recolhida em cada máquina após o término da aplicação. Isto corresponde a ter um "buffer infinito" nos serviços de traço em cada máquina. Só após a execução da aplicação terminar é que um monitor central requer o envio do traço coligido em cada representante do serviço local. A detecção do fim da aplicação pode ser determinada pela ferramenta, por exemplo, por eventos que lhe são entregues, indicando o término de cada processo da aplicação.

Com maior pormenor, cada serviço local de traço compreende os seguintes elementos:

- uma interface para controlar/parametrizar o seu próprio funcionamento (configuração);
- uma operação para inserir registos de eventos no repositório do serviço e outra operação para pedir os existentes (*Put*/*Get*);
- uma lista de subscritores consumidores, na qual fica registada a função de cada subscritor destinada à entrega dos registos;
- uma lista de subscritores produtores, onde uma função fica registada para obter os registos, que o produtor disponha;
- um canal de eventos associado, para propagar notificações urgentes a todos os clientes do serviço.

É possível usar este serviço num modo em que o consumidor "pede" a informação

quando pretendido; ou num modo em que o serviço toma a iniciativa de enviar para todos os subscritores consumidores a informação que recebe.

Serve assim, este serviço, de intermediário — transportador da informação — entre os produtores e os consumidores de registos. Estes modos procuram cobrir as mais variadas utilizações. Dependendo da utilização, os sensores e as ferramentas (ou outros serviços) tomam papéis activos ou passivos, no que diz respeito ao desencadear do transporte da informação pelo serviço de traço. Todos os padrões possíveis encontram-se na seguinte tabela, que são depois descritos:

	sens. activo	sens. passivo
ferramenta activa	1	2
ferramenta passiva	3	4

Estes padrões correspondem às seguintes situações:

1. os sensores injectam no serviço de traço local os registos que descrevem os eventos recolhidos. Os serviços de traço podem armazenar essa informação até que esta seja pedida explicitamente pelas ferramentas. Quando uma ferramenta pede a informação recolhida até esse instante, esta é-lhe então entregue. O meio de comunicação é aqui passivo para não perturbar a aplicação. A situação extrema é o caso da recolha *a posteriori* quando a ferramenta só vai pedir a informação após detectar o término da aplicação.

2. neste caso o sensor apenas se regista como produtor de informação. A ferramenta, tal como no caso anterior, desencadeia o pedido do processo de transferência. Este caso pode corresponder às situações em que os sensores coligem informação sob a forma de contadores ou indicadores que reflectem o estado da aplicação ou da máquina, os quais são consultados pela ferramenta durante a execução da aplicação (*on-line*) quando necessário.

3. nesta situação a ferramenta regista-se apenas como consumidora da informação. À medida que a informação é coligida pelos sensores e injectada no serviço de traço, esta vai fluindo até às ferramentas sem qualquer acção destas. Este caso contempla as situações de monitorização efectuada enquanto a aplicação se executa (*on-line*), casos em que as ferramentas vão consumindo a informação recolhida num fluxo contínuo.

4. este último caso revela-se inútil se não existir outra qualquer entidade (serviço ou ferramenta) que desencadeie o processo de transferência da informação, já que tanto os produtores como os consumidores estão inactivos. Nesta situação podemos considerar, por exemplo, os casos em que a recolha de informação e o respectivo tratamento são despoletados periódicamente, com base em intervalos de tempo (a entidade que desencadeia o processo é aquela que controla esses intervalos de tempo).

Na situações em que se pretende uma utilização do traço apenas após o termino da aplicação, será necessário que a ferramenta, quando tal decida, desencadeie explicitamente o processo de transferência completa desse traço para a ferramenta. Para tal, será conveniente uma primitiva para requerer explicitamente o envio de todo o traço coligido (tipo *flush*). Esta desencadeia um modo de funcionamento como na situação 3, onde cada serviço passa a informação disponível aos respectivos subscritores.

Vejamos a seguir as operações abstractas a suportar pelo serviço de traço:

SetWorkingParam(long buffsize) — apenas vamos considerar um parâmetro para configurar o seu funcionamento. Este define o tamanho do repositório interno, que pode variar de zero (difusão imediata dos registos entrados) até infinito (os registos são guardados, podendo recorrer a um ficheiro local), até que sejam pedidos pelos consumidores;

SetProd(produce) — o produtor regista junto do serviço a sua função de produção. Quando requerido pelos consumidores o serviço invoca a função registada para que esta lhe entregue os registos e depois passa essa informação aos consumidores;

SetCons(consume) — o consumidor subscreve o serviço indicando uma função responsável por receber a informação. Quando necessário, e dependendo da configuração, o serviço invoca essas funções entregando-lhes os registos existentes;

Put(register) — quando existe um novo registo este é explicitamente introduzido no serviço usando esta operação;

Get(register) — quando interessado, o consumidor pede explicitamente ao serviço todos os registos existentes invocando uma operação para esse efeito;

Flush() — esta operação permite que o consumidor, quando interessado, peça explicitamente ao serviço todos os registos existentes.

EvCh GetEvCh() — obter o identificador do canal de eventos associado ao serviço. Através deste, podem os sensores/actuadores, enviar, eventualmente, notificações aos clientes do serviço; os clientes podem subscrever este canal para receberem essas notificações.

A utilização deste serviço pode revelar-se complexa, especialmente no caso deste dever ser acedido concorrentemente por mais de uma ferramenta, em particular se emitirem comandos de configuração incompatíveis. Deve-se então optar por construir sobre este, um outro serviço vocacionado para a gestão da monitorização, de modo que funcione como coordenador global para as várias ferramentas. Fica este responsável pelo lançamento e configuração de uma infraestrutura construída com os serviços de traço, de acordo com a configuração pretendida (e arquitectura sob monitorização), servindo também de intermediário e coordenador de todas as ferramentas que acedam a estes serviços de traço.

4.7.2 Serviço de controlo

Como exemplo de um serviço de controlo, apresenta-se aqui o desenho de um vocacionado para a depuração. Este serviço baseia-se em trabalho antes desenvolvido com base no primeiro protótipo da DAMS, para a depuração de aplicações paralelas e distribuídas. Este foi usado para ensaiar a necessidade de uma infraestrutura geral que permitisse desenvolver outras aplicações com base num conjunto de funcionalidades comuns aos depuradores sequenciais. Seria assim mais fácil a construção de novas ferramentas que tivessem necessidade de aceder aos processos da aplicação distribuída, p.e. para observar o seu estado (dados, fluxo de execução, etc.) ou controlar a sua execução (ordem de execução entre os vários processos, p.e.).

Nesse sentido foi desenvolvido um sistema dedicado a essa infraestrutura para depuração (DDBG [25, 35]), e outro semelhante (o PDBG[81, 35]), mas em que este último utiliza como base a primeira versão do sistema DAMS. As soluções desenvolvidas nesse projecto permitiram verificar, para o segundo caso, a vantagem de se dispor de uma infraestrutura de base, sobre o qual as várias ferramentas foram na altura desenvolvidas [34, 22, 23].

Partindo dessa experiência, veja-se como se pode dispor que um serviço semelhante a ser agora suportado pela nova DAMS, continuando a permitir assim a integração deste tipo de operações na DAMS. Nesta secção apresenta-se um serviço básico de controlo, enquanto na

secção 6.4 será discutida a implementação deste serviço, e um outro para depuração distribuída, equivalente a outro desenvolvido na primeira DAMS.

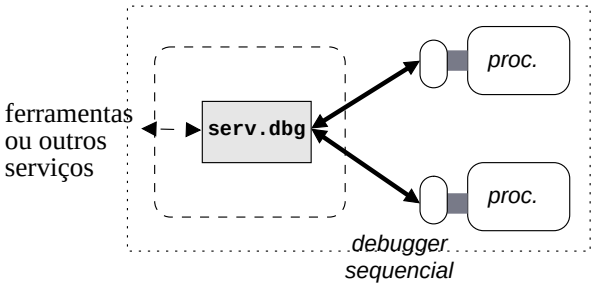

Figura 4.14: Um serviço para depuração na DAMS

Este será um serviço local, e responsável por disponibilizar em cada máquina, um conjunto de comandos para controlo dos processos, com vista à sua depuração, a partir de qualquer outra máquina na DAMS. Estas operações correspondem a tornar remotamente acessíveis as operações habitualmente suportadas pelos tradicionais depuradores sequenciais (*debuggers* como p.e. `gdb`), sendo em princípio, através da interacção com estes depuradores que se podem assim controlar os processos presentes em cada máquina (ver figura 4.14). Este é um exemplo da situação em que um actuador (neste caso o depurador sequencial) é integrado na infraestrutura DAMS por meio de um serviço que lhe serve de interface ou intermediário.

Pode-se, à partida, identificar dois tipos de interacções entre as ferramentas ou outros serviços e este serviço para controlo. Por um lado, a invocação de operações para actuar sobre os processos da aplicação, como por exemplo: parar, continuar, colocar um ponto de paragem[7], etc. Por outro lado a necessidade de passar informação à entidade cliente, sobre alterações no estado das computações controladas, como seja: parou num ponto de paragem. O primeiro aspecto é suportado pela interface oferecida pelo serviço que realiza uma API com todas as operações a suportar. O segundo aspecto é realizado recorrendo ao mecanismo de canais de eventos. Para este último, cada instância do serviço em cada máquina, será responsável pela criação de um canal de eventos próprio, através do qual propaga notificações de todas as alterações assíncronas detectadas pelos depuradores sequenciais. A entidade cliente pode, em resposta às notificações recebidas, actualizar o seu estado interno, ou a informação oferecida ao utilizador final, no caso de uma ferramenta.

[7] Em Inglês: *breakpoint*.

Várias ferramentas podem tirar partido desta infraestrutura e destes comandos, controlando os vários processos. Por outro lado, estas ferramentas podem mais facilmente ser usadas em aplicações que se executam sobre outras plataformas, desde que a interface oferecida pelo serviço na infraestrutura DAMS se mantenha inalterada.

Como a DAMS permite que vários clientes (logo, ferramentas) acedam a cada serviço, temos a partilha do estado de cada processo sob o controlo deste serviço entre os vários clientes. Tirando partido do mecanismo de difusão de eventos, todos os clientes podem receber "notificações" de alterações relevantes no estado da computação, sendo estas automaticamente propagadas aos vários clientes (p.e. para sincronizarem as respectivas visões do estado da aplicação, de cada vez que um processo pára ao atingir um ponto de paragem).

As operações a considerar são:

Attach(pid) — permite a ligação a um processo específico. Corresponde efectuar as inicializações necessárias para que o referido processo possa passar a ser controlado;

Detach(pid) — permite libertar o processo indicado do controlo do serviço;

Stop(pid) — pede a paragem da execução do processo indicado;

Step(pid) — avança uma instrução no processo indicado (préviamente parado);

Continue(pid) — continua a execução do processo, que antes estava parado;

SetBreakpoint(pid, breakpoint) — coloca um pondo de paragem no processo indicado. O *breakpoint* tem a indicação do ficheiro fonte e linha onde o programa deve parar.

DelBreak(pid, breakpoint) — remove o ponto de paragem indicado;

GetBacktrace(pid) — obter a lista das funções activadas na pilha de execução (*backtrace*);

GetValue(pid, var) — obter o valor da variável indicada;

SetValue(pid, var, value) — alterar o valor da variável indicada.

Além destas, existe ainda a possibilidade de requerer acesso a um canal de eventos específico, através do qual este serviço anuncia acontecimentos assíncronos que ocorrem nos processos sob o seu controlo:

EvCh GetEvCh() — obter o identificador do canal de eventos deste serviço.

Através deste canal prevêem-se as notificações na alteração do estado da execução da computação.

Este tipo de controlo é suficientemente poderoso para permitir a implementação de outro tipo de ferramentas. Este tipo de funcionalidades abrangem os requisitos das usadas no desenvolvimento de sistemas para *steering* [124], *replay* de aplicação distribuídas [82], teste sistemático da aplicação [84, 34, 35], etc...

4.8 Conclusão

Neste capítulo foi proposto um modelo e uma arquitectura abstracta de suporte às actividades de monitorização e controlo de aplicações distribuídas. O modelo propõe uma organização flexível e extensível, baseada num núcleo sobre o qual se executam o conjunto aberto de serviços, que realizam as funcionalidades específicas a cada cenário de monitorização. Cabe aos serviços possibilitar, dependendo de cada caso em particular, o modo como as ferramentas podem tirar partido das funcionalidades oferecidas e como as podem partilhar. Por outro lado, procura-se facilitar o desenvolvimento das ferramentas, assim como disponibilizá-las nos mais variados ambientes de aplicações paralelas e distribuídas.

Capítulo 5

Implementação da arquitectura DAMS

Neste capítulo são descritas as dimensões da implementação de infraestruturas que suportam o modelo da DAMS. São também referidas as abordagens seguidas na implementação de dois protótipos sobre duas plataformas computacionais diferentes.

5.1 Introdução

Como já foi referido, um primeiro protótipo do modelo DAMS foi desenvolvido, sendo a partir deste que a vantagem de um sistema deste tipo foi reconhecida e que se concebeu então o modelo apresentado no capítulo anterior. Estes trabalhos podem ser resumidos nos seguintes pontos:

1. um primeiro protótipo já referido, implementado sobre a plataforma PVM, que permitiu verificar a vantagem de uma plataforma comum no desenvolvimento de diversas ferramentas, que permitiu realizar diversas experiências da sua utilização;

2. um segundo protótipo, implementado pelo autor com base no modelo proposto no capítulo 4, e o principal tema deste capítulo. Além de manter as vantagens já reconhecidas no protótipo anterior, mantendo válidas as experiências antes realizadas (algumas foram trivialmente convertidas para este novo protótipo), permitiu a implementação de novos serviços e o ensaio de novas situações de aplicação.

Neste capítulo discutem-se os principais aspectos da implementação do protótipo segundo o novo modelo proposto para a DAMS, sobre uma plataforma Corba (no caso, a implementação ORBit). Esta plataforma revelou-se vantajosa para facilitar a implementação mas, dada a separação mantida entre o nível da plataforma e os níveis superiores da arquitectura da DAMS, outras plataformas poderão vir a ser usadas. É descrito também, o primeiro protótipo, onde a plataforma usada foi o PVM, discutindo-se os aspectos particulares da utilização desta plataforma.

5.2 Arquitectura da DAMS

Para a concepção de uma arquitectura que suporte a implementação, para uma plataforma concreta, de um sistema DAMS, de acordo com modelo proposto no capítulo 4, é necessário que esta implementação tenha acesso aos recursos dispersos pelos vários nós que se pretendem monitorizar. Esta distribuição implica, necessariamente, ter um representante do sistema DAMS em cada nó (ver fig. 5.1), que possa observar e controlar os processos da aplicação "alvo" e a plataforma que a suporta. Este representante irá incluir, no mínimo, entidades contendo os componentes de cada serviço que necessitam de acesso directo ao alvo da sua monitorização, via os sensores e actuadores existentes. Estas entidades desempenham o papel de "servidores de serviços", ao implementar as funcionalidades dos respectivos serviços que necessitam de interacção directa com os alvos da monitorização, aí residentes. Outra responsabilidade é também, permitir a partilha de estado ou recursos, entre vários clientes.

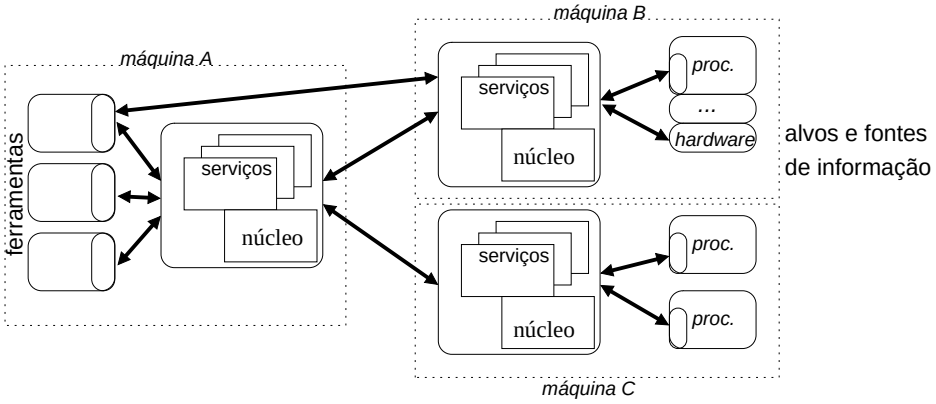

Figura 5.1: Arquitectura da DAMS

As entidades existentes em cada representante da DAMS são acessíveis remotamente,

para que qualquer entidade possa requerer as operações das entidades presentes neste representante. Estes pedidos são realizados, forçosamente, sobre a plataforma de comunicação que liga os vários nós, segundo um modelo cliente/servidor ou segundo um mecanismo de troca de eventos, sobre uma camada abstracta equivalente à referida na secção 4.6.4.

Em cada nó é necessário gerir quais os serviços localmente oferecidos e torná-los acessíveis às restantes entidades. Esta é uma responsabilidade do núcleo DAMS, que também inclui no representante de cada nó, a implementação necessária, para:

- gerir o registo dos servidores locais disponíveis nesse nó, assim como de quaisquer outros serviços aí implementados, permitindo a sua descoberta e utilização pelas entidades, locais ou remotas, que deles necessitem;
- gerir os canais de eventos aí criados, permitindo a propagação de cada evento pelos respectivos subscritores, mesmo se remotos;
- permitir a gestão das máquinas no sistema DAMS, assim como as restantes funcionalidades, oferecidas pela plataforma local, necessárias às operações anteriores;
- suportar, sobre a plataforma de comunicação existente, uma infraestrutura que permita as interacções entre as entidades, atrás referidas.

Estes representantes distribuem-se por vários nós, definindo a máquina virtual DAMS, distribuindo as funcionalidades do sistema DAMS por todas as máquinas monitorizáveis. As ferramentas (ou quaisquer outras aplicações) que interagem com a infraestrutura DAMS, são clientes de um (ou vários) destes representantes, para aceder às entidades aí residentes. Por questões de eficiência, as ferramentas devem contactar tipicamente, o representante mais "perto", ou seja, se possível o representante residente no mesmo nó, mas podem no entanto, e quando necessário, ter acesso directo a entidades servidoras residentes nos representantes remotos. Tal será o caso em que a operação pretendida só pode ser executada pela entidade remota (p.e. a ferramenta quer monitorizar um processo que se executa nesse nó remoto).

5.2.1 Entidades

Na arquitectura distribuída sobre a qual o sistema DAMS se executa, as várias entidades abstractas identificadas no modelo DAMS, encontram-se também distribuídas, e realizadas

com base nos servidores da arquitectura de implementação, atrás referida, sendo que:

Serviços. A arquitectura típica dos serviços compreende duas partes ou componentes, vistos na secção 4.5. Uma API para acesso ao serviço, residente na entidade cliente e que oferece as funcionalidades específicas do serviço. No entanto, a realização desta API necessita, para algumas funcionalidades, de efectuar pedidos a um componente servidor que gere o estado interno do serviço, ou que interage com os recursos observados ou controlados pelo serviço. Permite também, se pretendido pelo serviço e dependendo da sua arquitectura interna, a sua utilização concorrente por vários clientes. Este servidor fará parte do representante da DAMS. Um exemplo particular destes servidores, são os que servem de interface com os sensores e actuadores, locais a cada nó, tornando-os conhecidos e acessíveis aos restantes serviços e às ferramentas que os pretendam utilizar.

Ferramentas. São entidades consumidoras da informação recolhida, ou controladores da aplicação, que não oferecem funcionalidades às restantes entidades. Estas ferramentas são os clientes finais dos serviços, assim como do núcleo DAMS. As interfaces de programação (API) que utilizam, oferecem um novo nível de abstracção sobre as funcionalidades realmente exportadas pelos servidores dos serviços. Por exemplo, podem esconder a arquitectura real do serviço (como no caso de no início da interacção com o serviço, poder existir uma negociação interna ao serviço, em que por questões de distribuição de carga, distribuição espacial, ou outra, este é reencaminhado para outro servidor, que vai realmente atender os pedidos seguintes).

O núcleo DAMS, compreende uma arquitectura idêntica à dos serviços, tendo no entanto de se basear na plataforma sobre a qual se executa para suportar as restantes entidades. Na sua API, utilizada por todas as entidades, encontra-se o suporte à implementação das interacções necessárias à implementação do modelo cliente/servidor, à propagação dos eventos, sua recepção e tratamento. O seu componente servidor suporta as operações de gestão previstas, assim como a implementação dos canais de eventos. Este suporte necessita de estar disponível em cada nó sob a monitorização ou controlo da DAMS. A sua disponibilidade num nó é garantida quando este é acrescentado à máquina virtual DAMS, via a primitiva `damsAddHost`. Esta chamada desencadeia os mecanismos para que o representante, antes referido, passe a executar-se também nesse nó.

Será por intermédio de um dos servidores que implementa o núcleo da DAMS, que as ferramentas acedem às funcionalidades desse núcleo, permitindo-lhes localizarem e acederem aos vários serviços. Ou seja, existe sempre este primeiro contacto a partir do qual os restantes serviços e respectivos servidores serão descobertos, de acordo com os requisitos do cliente. Os vários serviços serão também clientes deste núcleo, enquanto utilizadores das funcionalidades da DAMS, incluindo o seu registo, assim como para a localização de quaisquer outros serviços de que sejam clientes.

5.2.2 Interface com a plataforma

Dada a variedade de plataformas disponíveis para a computação paralela e distribuída, procurou-se escolher, para a realização do protótipo, uma que facilitasse a experimentação e o teste da funcionalidade do modelo.

A independência requerida face à plataforma usada, para cumprir os objectivos enunciados na secção 4.3, levam a uma realização da infraestrutura DAMS que permita a independência dos serviços e ferramentas, relativamente a essa plataforma. Evita-se também, na realização do núcleo DAMS, tirar partido de facilidades particulares da plataforma que serve de base à implementação deste protótipo. Pode-se, assim, dispor de implementações de instâncias da DAMS utilizando plataformas distintas, sem alteração dos serviços ou seus clientes (excepto nas interacções que estes tenham directamente com a plataforma, como com os sensores ou os actuadores usados).

A interface da DAMS com a plataforma cobre dois tipos de requisitos: suportar as funcionalidades do núcleo DAMS, em particular na gestão dos recursos presentes na plataforma (máquinas físicas e processos); por outro lado, suportar a interacção das entidades com o núcleo, assim como entre si. Nestas interacções, toma-se como ponto de partida o suporte abstracto referido no capítulo 4. Cada realização da DAMS exige a construção de *stubs*/*skels*, com as funcionalidades características dos modelos do tipo RPC, sobre a plataforma disponível. A realização do núcleo DAMS (ver fig. 5.2), também se baseia nesta interface, na implementação da sua API em cada entidade, por forma a permitir a interacção com o seu servidor.

O suporte para o núcleo da DAMS gerir as máquinas e os processos, pode ser implementado recorrendo ao sistema de operação, ou outras facilidades da plataforma (ver `dams-low`

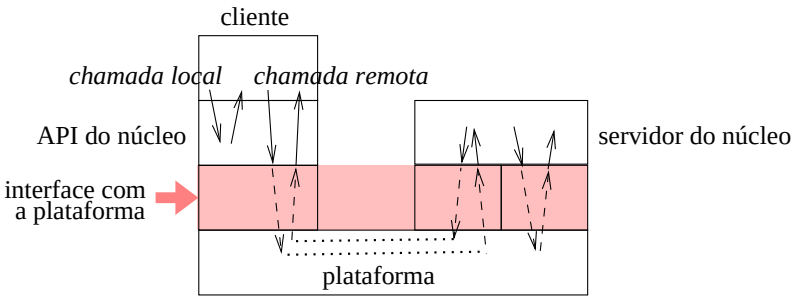

Figura 5.2: Realização do núcleo DAMS

na secção 5.5.2). No núcleo a DAMS, deve manter-se a informação de quais as máquinas participantes e quais os serviços disponíveis em cada uma.

Note-se que algumas operações da API cliente do núcleo, assim como no caso das API dos serviços, correspondem a chamadas locais ao próprio processo cliente, afectando apenas o estado desse cliente, sendo assim totalmente suportadas no processo cliente; outras, normalmente, desencadeiam alterações no estado global da DAMS ou do serviço, necessitando assim de interagir com outras entidades nos representantes da DAMS, possivelmente em processos remotos, onde esse estado é gerido; noutros casos, a funcionalidade pretendida só pode ser implementada remotamente, por exemplo quando um serviço necessita de aceder a determinado sensor ou actuador.

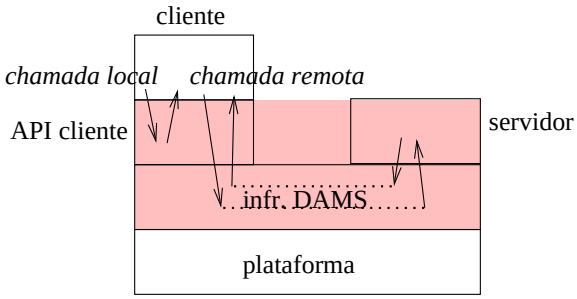

Figura 5.3: Implementação das interacções em serviços da DAMS

Para as interacções entre as entidades, o suporte necessário apoia-se nos mecanismos oferecidos pela infraestrutura DAMS (fig. 5.3), para a troca de informação que permite a implementação das interacções, independentemente da plataforma. A infraestrutura DAMS não pode assumir mais funcionalidades do que as previstas no modelo, sob pena de ficar dependente de sistemas de comunicação particulares, quando na abordagem DAMS se pretende

manter essa independência.

Comparação de alternativas

Considerou-se que os protótipos, em particular, suportariam apenas a execução em ambientes do tipo UNIX (Linux neste caso), sendo, por isso, a plataforma usada na sua implementação, escolhida de entre as disponíveis nesse sistema. Neste ambiente, dispomos de várias plataformas que podem ser usadas para a realização de um protótipo. Verificando como os requisitos da DAMS podem ser satisfeitos, podemos avaliar as diferenças entre as plataformas e as exigências para a implementação da infraestrutura DAMS em cada uma. Os aspectos a considerar resultam da realização dos grupos de funcionalidades, apresentados para o modelo DAMS no capítulo anterior:

1. infraestrutura de comunicação;
2. gestão de máquinas;
3. gestão de serviços;
4. canais de eventos.

Referem-se em seguida, as plataformas genéricas que foram consideradas, representativas de vários níveis de abstracção e procuram-se avaliar as possibilidades da sua utilização como plataformas de suporte da arquitectura DAMS:

- interface do sistema (Unix/Posix [107]) — interface tradicional do sistema de operação UNIX, onde para o estabelecimento das interacções, se recorre à interface de *sockets*;
- modelos baseados em troca de mensagens, PVM [46] e MPI [101, 102] — mecanismos básicos de comunicação por mensagens entre processos, mesmo que distribuídos, com codificação de tipos de dados;
- invocação de procedimentos/métodos remotos, RPC *standard* [118] ou Corba [106] — mecanismos que suportam a criação de programas com chamadas a funções de processos diferentes, de forma transparente para o programador.

Interface UNIX

Dado esta interface ser a de mais baixo nível aqui analisada, não surpreende que pouco tenha para oferecer nos vários aspectos pretendidos pelo modelo DAMS. Esta plataforma tem a vantagem de ser uma norma nos sistemas que estamos a considerar e de possibilitar uma implementação específica e mais eficiente da DAMS (incluindo no uso de recursos), sem depender de qualquer outro sistema.

Usando a interface de programação por *sockets* para as interacções, a infraestrutura DAMS terá ela própria de disponibilizar as operações necessárias e os protocolos para a codificação dos diferentes tipos de dados dos pedidos, para implementar os *skels* e *stubs* necessários, referidos na secção 4.6.4. Incluirá também a implementação dos aspectos de identificação, registo das entidades e sua localização, assim como o encaminhamento dos pedidos, sua recepção e despacho para a entidade respectiva e retorno das respostas. Quanto à entrega dos eventos aos subscritores dos canais de eventos, podem ser usados mecanismos como sinais, *threads* dedicados à recepção desses eventos ou utilização dos mecanismos do tipo *select* ou *poll*, para testar a chegada desses eventos.

A gestão de máquinas na DAMS, pode ser realizada com o recurso às facilidades comuns nos sistema de operação, nomeadamente, aos serviços de rede que permitam a sua nomeação e identificação (p.ex. DNS[1]), e aos que possibilitam o lançamento remoto de processos (p.ex. SSH[2]).

Interfaces PVM e MPI

Estas plataformas oferecem já várias abstracções que vão ao encontro dos requisitos indicados para a realização da DAMS. Em particular, suportam uma noção de máquina virtual, que permite ao sistema aceder a todas essas máquinas e identificar globalmente qualquer processo (*task*). A gestão de máquinas na DAMS pode assim facilmente ser realizada com base na máquina virtual do PVM ou do MPI (a gestão dinâmica, nesta última, só é possível sobre o MPI-2).

A comunicação por mensagens suporta um protocolo que permite a troca de diferentes

[1] *Domain Name System*.
[2] *Secure SHell*.

tipos de dados, mesmo entre arquitecturas diferentes, sendo a sua conversão e serialização asseguradas pela plataforma. Ambas as plataformas oferecem primitivas para troca de mensagens, bloqueantes e não bloqueantes, como sejam: `MPI_Send()` / `pvm_send()`, `MPI_Recv()` / `pvm_recv()`; e para a codificação dos dados nas mensagens: `MPI_Pack()` / `pvm_packf()` e `MPI_Unpack()` / `pvm_unpackf()`. Estas primitivas podem ser utilizadas na implementação dos pedidos de operações, de forma idêntica às referidas na secção 4.6.4. No entanto, tem de ser desenvolvido o protocolo para a codificação de pedidos e respostas, assim como para a indicação de qual a entidade destino dentro de cada processo.

Cada processo (*task*) tem de dispor, no seu programa, de um ponto de entrada onde recebe as mensagens, descodifica-as e encaminha-as para a entidade respectiva, de acordo com o seu conteúdo, executando um ciclo permanente de recepção de mensagens e seu tratamento.

Quanto aos canais de eventos, o envio de mensagens não bloqueantes está previsto e é oferecido pelas plataformas. No que diz respeito à recepção das notificações, estamos perante uma situação idêntica à dos *sockets*. A maneira de lidar e atender as entregas de notificações em qualquer instante, tem de ser implementada utilizando as facilidades oferecidas pela plataforma para testar a chegada de mensagens (no caso do PVM, pode-se também tirar partido do mecanismo de sinais que se encontra estendido para enviar sinais UNIX a qualquer processo na máquina virtual).

Note-se, no entanto, que estas plataformas são habitualmente usadas como modelo de programação paralela, pelas aplicações alvo do sistema de monitorização. Isto coloca o problema de a máquina virtual ser partilhada e, como tal, também o respectivo espaço de identificadores de processos. Não podemos monitorizar nós que ainda não pertençam à máquina virtual da aplicação, e é necessário considerar as possíveis interferências entre a aplicação e os processos adicionais da arquitectura DAMS. Para a utilização da DAMS com uma qualquer aplicação, será necessário garantir que os algoritmos e as convenções adoptadas na aplicação, não assumem que todos os processos na máquina virtual fazem parte da aplicação, o que muitas vezes exige a alteração do seu programa. Mesmo assim, o sistema PVM pôde ser usado como plataforma de suporte no primeiro protótipo da DAMS, como é apresentado mais à frente.

Interfaces RPC e Corba

Estes sistemas, oferecem de base, os mecanismos para a criação de *stubs* e *skels*, incluindo os protocolos de chamada e resposta, assim como de codificação dos vários tipos de dados. Internamente, primitivas semelhantes às referidas na secção anterior, implementam os *stubs* e *skels*. Linguagens de descrição das interfaces das operações implementadas permitem a geração automática desse código. Torna-se assim bastante atractivo o seu uso, para a infraestrutura de comunicação e para implementar as interfaces dos servidores dos serviços. O uso de um sistema destes, como ponto de partida, permite um mais rápido desenvolvimento do novo protótipo.

Mantendo os pressupostos assumidos, de não ficar dependente de facilidades específicas destes sistemas, como apresentado na secção 4.6.4, pretende-se suportar apenas as seguintes funcionalidades:

1. interacções cliente/servidor para a realização de operações síncronas internas aos componentes dos serviços e do núcleo;

2. interacções cliente/servidor para a entrega de eventos, ou seja, operações sem valores de retorno e que não necessitam de bloquear o produtor do evento;

3. registo de funções para a recepção dos eventos e o mecanismo para receber estes eventos e invocar as funções associadas.

Estas plataformas cobrem as necessidades de comunicação, mas não oferecem funcionalidades que suportem a gestão da máquina virtual DAMS. A infraestrutura DAMS deve então tirar partido do próprio sistema de operação, nas restantes funcionalidades (como referido para o UNIX) e manter a informação de quais as máquinas participantes.

Quanto à gestão de serviços, poder-se-iam utilizar as facilidades para registo dos servidores nestas plataformas (`portmap` nos RPC e `nameservice` no Corba), mas tendo em conta o tipo de registo requerido, revela-se mais simples criar, no servidor que suporta o núcleo DAMS, estas funcionalidades. A implementação da infraestrutura DAMS fica assim, apenas dependente da implementação dos *stubs* e *skels* para cada plataforma, sem comprometimento com estes serviços específicos.

Para a realização dos canais de eventos, no caso do Corba, está prevista que uma chamada

possa não bloquear, quando um método não retorna qualquer valor e é declarado `oneway`. No caso dos RPC, situação semelhante pode ser conseguida por alteração do *stub* gerado. Quanto à entrega das notificações, poderemos usar a plataforma para tornar cada cliente também um servidor, passível de ser chamado para a entrega das notificações. Neste último caso, podemos disponibilizar primitivas de teste e despacho desses pedidos, para uso pelo programador do cliente.

Conclusão

Neste trabalho considerou-se que seria mais importante o teste prático dos conceitos introduzidos e a avaliação funcional do sistema DAMS em ambiente experimental, delegando os problemas de desempenho e uso de recursos para desenvolvimentos futuros. Assim, o recurso a um sistema tipo RPC ou Corba, oferecido pela plataforma de base como ponto de partida, permite um mais rápido desenvolvimento do protótipo, desde que a possível penalização no desempenho e no uso da banda passante da infraestrutura física não sejam impeditivos e que a infraestrutura DAMS assim como os serviços e clientes, não fiquem comprometidas com essa plataforma.

A implementação CORBA usada (ORBit) efectua optimizações para tirar partido das situações em que a comunicação se processa entre entidades na mesma máquina —usa uma zona de memória partilhada para efectuar a comunicação de forma mais eficiente— ou, quando no mesmo processo (o mesmo espaço de endereçamento) —efectua as chamadas directamente entre as funções— minimizando a penalização sobre o desempenho. Só nas situações de objectos remotos, recorre aos protocolos de rede (TCP/IP). Estes detalhes estão a um nível inferior, e completamente abstraídos da própria infraestrutura DAMS.

A abordagem de implementar de raiz toda a infraestrutura de comunicação foi, pelas razões anteriores, posta de parte. A utilização dos sistemas PVM ou MPI não foi considerada para o segundo protótipo, porque, as suas funcionalidades e limitações para suportar a DAMS-1 já haviam sido exploradas no primeiro protótipo. Quer por não oferecer a criação automática das funções de *stub* e *skel*, e o mecanismo de atendimento dos pedidos no lado servidor. Mais ainda, pelo conflito que o uso destas plataformas poderia ter com as próprias aplicações.

No caso do modelo CORBA e da sua linguagem IDL (*Interface Definition Language*), é

bastante conveniente fazer a correspondência da interface de cada servidor de serviço (ou do núcleo DAMS) em *interfaces* IDL e o envio de eventos por chamadas `oneway`. Exige-se a implementação de um processo servidor que realize a entidade servidora do núcleo DAMS, que será a base do representante da DAMS em cada nó da máquina virtual DAMS. Esta foi a escolha para o segundo protótipo, em particular o sistema ORBit, já disponível nas máquinas de teste.

5.3 O primeiro protótipo sobre o PVM

O primeiro protótipo de um sistema DAMS, apresentado na secção 4.2, foi desenvolvido sobre o sistema PVM, apresentando as seguintes características:

- a máquina virtual para uso da DAMS é suportada sobre a do PVM;
- os vários processos no sistema DAMS — ferramentas, *Service Manager*, *Local Manager* e *drivers* — são realizados como *tasks* PVM;
- as interacções entre os diferentes processos assentam nas primitivas de comunicação por mensagens do PVM;
- cada *Service Module* é suportado por um *thread* de sistema, no interior do processo SM (*Service Manager*).

A API de cada serviço é implementada sobre *stubs*, programados para o efeito, que interagem com o respectivo *Service Module*, efectuando pedidos, sobre um protocolo assente em primitivas tipo *pack*, *unpack*, *send* e *recv*, semelhantes às equivalentes do PVM. Estas primitivas são disponibilizadas pela DAMS e acrescentam à interface do PVM, a identificação do serviço a que dizem respeito, assim como da operação que está a ser pedida.

Cada *Service Module* é implementado por um *thread*, criado no arranque do SM, ficando associado ao identificador do respectivo serviço. Cada pedido vindo de uma ferramenta, chegado ao SM, é por este encaminhado para o *thread* do SMod identificado com o serviço, sendo colocado numa fila de pedidos que o SMod vai processando sequencialmente.

As verdadeiras interacções com os processos alvo, são efectuadas pelos *drivers* de cada serviço, por pedidos que lhes são dirigidos pelos respectivos SMod, via LM. Estes *drivers* são

geridos pelo respectivo SMod utilizando a infraestrutura da DAMS. Sempre que um serviço necessite de aceder a determinada máquina, deve o seu SMod pedir ao SM o lançamento do seu *driver* no nó pretendido, sendo esse pedido executado pelo LM que executa no referido nó, com base nas primitivas do PVM. A partir daí, passa a ser possível ao SMod efectuar pedidos ao seu *driver* nesse nó.

Estes pedidos dos SMod para os respectivos *drivers*, são suportados por primitivas semelhantes às existentes para a implementação da API cliente, mas disponibilizadas pelo SM. Estas acedem à plataforma PVM, sendo no entanto garantida pelo SM a exclusão mútua entre pedidos concorrentes por parte dos vários *threads* (SMod). Sequencializam-se assim os pedidos, para garantir que não há anomalias no comportamento das primitivas PVM, visto que estas não são reentrantes.

Em resumo:

- todas as mensagens incluem o identificador do serviço a que dizem respeito; e existe um identificador interno, reservado para interacção de controlo entre o SM e LM, para, por exemplo, controlar o lançamento ou terminação dos *drivers* dos serviços;

- as mensagens incluem o pedido criado pelo SMod, necessário para identificar qual o pedido que é feito e respectivos dados;

- cada LM distribui os pedidos que lhe chegam vindos do SM, para o respectivo *driver*, de acordo com o identificador de serviço que vem no pedido; a resposta vinda do *driver*, segue um caminho inverso, do LM para o SM, que este último entrega ao *thread* do respectivo SMod, que aguarda a resposta.

Os eventos são realizados como mensagens, com origem nos *drivers*, encaminhadas para o respectivo SMod, onde ficam a aguardar que o cliente do serviço as requeira, por um pedido existente para esse efeito (*polling*).

Sobre este protótipo, foram desenvolvidos vários serviços, no âmbito de diferentes projectos. Utilizando estes serviços, foram implementadas diversas ferramentas, discutidas no capítulo 6.

5.4 O segundo protótipo da DAMS

Nesta secção e seguintes, discute-se a realização do segundo protótipo da DAMS, baseado no novo modelo proposto no capítulo anterior.

Para a realização do representante DAMS referido na secção 5.2, em cada máquina, são várias as possibilidades. Poder-se-á optar por um único processo que contém todas as entidades servidoras necessárias. Mas, dada a separação desejada entre serviços e destes em relação ao núcleo da DAMS, sendo as suas interacções baseadas apenas na infraestrutura de comunicação que identifica cada entidade, é também possível implementar cada uma destas em termos de processos autónomos. Outras alternativas se colocam, que podem ser mais adequadas, dependendo de cada serviço, e da arquitectura interna pretendida para cada serviço. Algumas alternativas que se podem considerar como exemplos:

um único fluxo de execução — um único processo que suporta a implementação do núcleo DAMS em cada nó bem como todos os servidores dos serviços suportados nesse nó. Como consequência, cada máquina dispõe de um sistema centralizado, onde os vários pedidos serão atendidos, sequencialmente. Esta organização tem a limitação de só se atender um pedido de cada vez, mesmo quando estes sejam destinados a serviços distintos. Esta situação é tanto mais grave quanto mais demorado for o atendimento da cada operação, aumentando este com o aumento na taxa de pedidos que lhe chega. Este problema pode comprometer o objectivo dos canais de eventos, já que o próprio servidor do núcleo, tal como os outros servidores, vai estar sujeito ao atendimento sequencial para a recepção de novos eventos e sua propagação. Por outro lado, fica simplificada a gestão de acessos concorrentes ao estado do serviço ou da aplicação monitorizada.

um fluxo de execução próprio a cada serviço — múltiplos fluxos de execução (implementados por *threads* ou por vários processos), onde cada um suporta um serviço. Permite concorrência entre os pedidos dirigidos a serviços diferentes assim como ao núcleo do sistema, garantindo que um pedido a um serviço, não bloqueia os pedidos para os restantes serviços. Continua a existir o problema no caso de pedidos simultâneos dirigidos ao mesmo servidor, mas continuam a evitar-se os problemas de concorrência dentro do serviço, já que só é atendido um pedido em cada instante. O atendimento de pedidos de operações por um serviço, mesmo demoradas, já não vai interferir com

os outros serviços nem com o núcleo DAMS. Pode exigir suporte da plataforma subjacente para a criação dos vários fluxos de execução, o que não é previsto pelo modelo da DAMS, e dependerá assim da sua implementação, em cada caso.

um fluxo de execução próprio a cada pedido — para cada pedido recebido, é criado dinâmicamente um novo *thread* (ou processo) para o atender, permitindo o atendimento de múltiplos pedidos simultâneos. Esta abordagem exige que cada serviço faça a gestão dos múltiplos fluxos em execução, podendo criar problemas com a partilha do estado do serviço e/ou da aplicação monitorizada, quando os vários fluxos lhes pretendem aceder. Neste caso, cada serviço tem de implementar, de acordo com as suas características e requisitos pretendidos, a gestão desses múltiplos fluxos e acesso aos seus recursos. Exige-se também suporte, por parte da plataforma, para a criação dos vários fluxos de execução e sua gestão, ficando para tal dependente da plataforma subjacente.

O modelo da arquitectura da DAMS permite qualquer abordagem para a realização de cada serviço, dado que a sua organização interna não está à partida definida, e se encontra escondida na API cliente. Na sua realização, a componente servidora é uma entidade que expõe apenas a sua interface, sendo a respectiva implementação transparente para a componente cliente. Na abordagem de um único processo, antes referida, a interface de cada serviço é acrescentada à interface oferecida pelo processo representante da DAMS, mas é garantida a separação do espaço de nomes e mantém-se o registo e identificação autónomos de cada serviço. A abordagem de um fluxo de execução por pedido, introduz uma complexidade na sua implementação, que pode não compensar o que se pretende ganhar na resposta aos pedidos simultâneos, mas pode ser conveniente para alguns serviços. A implementação de cada serviço definirá a sua organização interna e a gestão que faz dos vários fluxos e do acesso a recursos partilhados, de acordo com as suas necessidades.

É também possível a realização de sistemas com soluções mistas, onde cada serviço adopta a organização que melhor se adequa aos seus objectivos. Enquanto certos serviços implementam o respectivo servidor no mesmo processo que o do núcleo DAMS, outros serviços podem optar por dispor de servidores em processos dedicados, ou recorrer à plataforma de suporte para a criação dinâmica de múltiplos *threads*.

Nas organizações com um único fluxo de execução, os problemas de bloqueio do pedido enquanto um pedido é atendido, podem ser contornados na concepção do próprio serviço. Há que evitar pedidos passíveis de bloquear o servidor por longos períodos, o que poderá

ser conseguido por um protocolo entre API cliente e o respectivo servidor, onde a verdadeira resposta é diferida, sendo utilizado o mecanismo de eventos para permitir notificar o cliente, quando a resposta fica disponível.

No protótipo desenvolvido para o novo modelo da DAMS, existe um único processo em cada nó, o qual inclui a implementação do próprio núcleo DAMS, junto com os servidores que realizam os serviços. Este processo funciona como o representante da DAMS nesse nó, suportando todas as funcionalidades, e podendo as ferramentas executar em qualquer das máquinas no sistema DAMS. A razão da opção tomada prende-se com a simplicidade do desenvolvimento, de modo a permitir a experimentação das funcionalidades do modelo, face a diferentes cenários de utilização.

5.4.1 Interface com o ORBit

Dada a opção de se recorrer a uma implementação da norma Corba como plataforma de suporte à comunicação, no caso o ORBit, dispomos da possibilidade de especificar as interacções entre as várias entidades na linguagem IDL do modelo Corba, gerando depois as funções de interface para cliente (*stubs*) e para o servidor do serviço (*skels*). As primeiras são utilizadas pela implementação da API de interface no lado do cliente. As segundas, por sua vez, são ligadas respectivamente, às funções que implementam o serviço no lado servidor, quando este é criado junto do núcleo DAMS.

A organização destes níveis pode ser traduzida segundo a figura 5.4, onde se representa a situação de uma ferramenta cliente, de um serviço e do núcleo DAMS. As interacções são realizadas com base nos vários níveis de abstracção, onde os superiores se baseiam exclusivamente nas funcionalidades dos imediatamente inferiores.

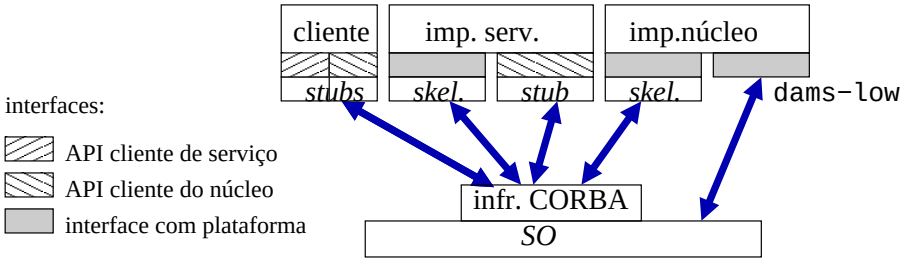

Figura 5.4: Níveis entre clientes e servidores de serviços

Para a implementação de cada serviço, na concepção da sua organização interna, o componente servidor é uma entidade realizada num objecto Corba, sendo necessário definir, numa "interface" IDL, as operações oferecidas. Os *stubs* e *skels* obtidos, não serão visíveis pelos clientes, nem nas implementações das respectivas operações nos servidores dos serviços, para procurar manter a sua independência.

Recorre-se a uma camada que serve de interface entre a implementação do serviço e as respectivas funções *skels* obtidas do IDL. Também a instanciação na infraestrutura CORBA (registo dos *skels* no Object Adapter) é efectuada por esta camada, no arranque do servidor. Esta interface trata da ligação das operações do servidor com o ORB, lidando com as identificações de objectos Corba, a ligação destes às respectivas implementações (via POA), gestão de memória e tipos para o ORB, e gestão do ambiente Corba (`CORBA_Environment`, para o tratamento de excepções/erros).

De forma idêntica, no lado dos clientes, a API oferecida ao cliente permite esconder destes a existência do nível CORBA. Junto com a API do núcleo DAMS, é incluída a implementação do sistema para o registo de funções para os *callbacks*, que permitem receber os eventos vindos do canal e a chamada do respectivo *handler*.

Uma segunda classe de interface aparece no caso do núcleo DAMS, a qual diz respeito ao suporte que é exigido à plataforma base, para obter informação sobre a máquina em que se executa, e para manipular os seus recursos recorrendo às funcionalidades oferecidas pelo sistema de operação, como sejam para desencadear a execução local ou remota de novos processos (esta camada, `dams-low`, é discutida na secção 5.5.2). O objectivo é tornar os serviços, na medida do possível, independentes da plataforma de suporte, e oferecer uma interface que facilite o transporte de toda a DAMS entre diversas plataformas.

5.5 Realização do núcleo DAMS

Cada cliente só pode interagir com a DAMS, após a inicialização do sistema, e da sua ligação ao núcleo DAMS. Para tal recorre à primitiva `damsInit()` que começa por inicializar e instanciar a interface CORBA do próprio processo invocador. Operação inversa existe para o desligar da DAMS e libertar os recursos usados. Estas primitivas são as seguintes:

`damsHost damsInit(int argc, char *argv[] )`

Inicialização do sistema de comunicação e estabelecimento da ligação com o sistema DAMS.

Uma vez activada a infraestrutura CORBA, passa-se a um processo de pesquisa do identificador do servidor do núcleo DAMS, com o qual se vai interactuar quando necessário (este identificador é um tipo opaco, específico da camada de comunicação usada; neste caso, será um identificador CORBA). Esta identificação é conseguida pela seguinte ordem:

1. esta é indicada como argumento;
2. esta é obtida de um repositório local à máquina. Cada instância do núcleo DAMS em cada máquina disponibiliza o seu identificador no ficheiro "/tmp/damsd.ior";
3. se nenhum identificador foi encontrado, uma instância do núcleo da DAMS é executada, obtendo o respectivo identificador pelo respectivo canal *stdout*, desse processo.

O identificador obtido, depois de validado (efectuada a ligação), é retornado ao invocador. Caso não tenha conseguido obter um identificador válido, é retornado um identificador nulo. Antes de retornar, é iniciado o mecanismo que permite a recepção de eventos. Para tal, é criado um servidor interno que pode receber pedidos de entrega de eventos.

`void damsEnd()`

Termina todas as ligações estabelecidas com o sistema DAMS e desactiva a recepção de eventos. Em particular, será terminada a sessão com o ORBit (`CORBA_ORB_shutdown`) e libertados os recursos usados pelo mecanismo de recepção dos eventos.

As restantes operações do núcleo, podem recorrer ao servidor usando o identificador obtido pela primitiva `damsInit`, para a realização da operação correspondente.

5.5.1 Implementação das operações do núcleo DAMS

A maior parte das operações do núcleo DAMS são realmente efectuadas por um conjunto de processos distribuídos pela máquina virtual DAMS, que gerem o estado do sistema. Na realidade, neste protótipo, cada processo representante da DAMS, mantém o seu próprio

estado e não estão implementados quaisquer suportes para a difusão desta informação ou tentativa de manter um estado global coerente[3]. Assim, cada um destes processos é uma instância do núcleo DAMS, que gere as entidades presentes na máquina e permite a gestão e localização destas, pelas restantes entidades remotas.

5.5.2 Gestão de máquinas

Cada máquina é identificada por um nome global (tipicamente, para redes IP, será o nome registado no DNS). O conjunto de máquinas (ou nós) que representa a máquina virtual DAMS, pode ser aumentado ou diminuído por recurso às funcionalidades dos servidores DAMS de cada máquina. No actual protótipo não é garantida a difusão e a actualização automática desta informação por todas as máquinas. Caso uma máquina funcione como gestor central, a partir da qual todas as outras são acrescentadas e removidas da máquina virtual, esta é a única que possui a visão global da máquina virtual.

Para as operações de gestão, é utilizado um módulo genérico (`dams-keys`) que implementa tabelas de pares <*chave,valor*>. Este módulo suporta as tradicionais operações para inserção, remoção, busca por chave e listagem de todos os pares. Uma destas tabelas é usada para manter a lista das máquinas, onde figuram os pares: <*nome de máquina, identificador CORBA do respectivo núcleo DAMS*>.

No atendimento dos pedidos para alteração da máquina virtual DAMS, o servidor do núcleo a quem foi dirigido o pedido recorre à plataforma para executar as operações e actualiza convenientemente a tabela de máquinas. A interacção com a plataforma é feita por um módulo de interface de nome `dams-low` (ver fig. 5.4), que abstrai dos detalhes dessa plataforma. Esta interface suporta as operações que permitem, ao servidor do núcleo DAMS, realizar as seguintes funções da API:

`damsHost damsAddHost( damsHost dh, char *hostname )`

pedir a adição de um nó físico (máquina) ao sistema DAMS. Este processo exige que o núcleo da DAMS recorra às facilidades da plataforma subjacente para pedir a execução de uma nova instância do representante da DAMS, no nó cujo nome foi indicado. É devolvido o identificador dessa nova instância nesse nó.

[3]Considerou-se que essa problemática ultrapassa o âmbito deste trabalho e vai além dos seus objectivos. No entanto, podemos admitir que essa funcionalidade poderá ser acrescentada à DAMS como um novo serviço.

`void damsDie( damsHost host )`

pedir ao nó indicado que termine. Todos os serviços suportados são também terminados, assim como a própria instância do servidor do núcleo DAMS.

`void damsDelHost( damsHost host, damsHost toremove )`

pedir para remover, do sistema DAMS, o nó cujo identificador é indicado (equivale a pedir ao "`toremove`" a operação `damsDie`).

`damsHostList *damsListHosts( damsHost host )`

pedir a lista de identificadores dos nós conhecidos desta máquina (o `host`).

`damsHost damsFindHost( damsHost dh, char *hostname )`

obter a identificação do nó cujo nome é indicado, por busca na tabela dos nós instanciados na máquina virtual da DAMS.

`procInfoList *damsListProcs( damsHost host )`

pedir a lista dos processos que se executam no nó. No protótipo corrente, é devolvido um vector contendo, para cada processo, o respectivo identificador local (`pid`), o identificador do respectivo utilizador (`uid`) e o nome do ficheiro executável.

`hostInfo *damsHostInfo( damsHost host )`

obter informação sobre o nó. No protótipo corrente, apenas devolve o respectivo nome no DNS.

Cada máquina é adicionada à máquina virtual DAMS, através da iniciação, nessa máquina do processo que inclui a entidade servidora do núcleo DAMS (e representante da DAMS). O identificador no CORBA, do servidor do núcleo, servirá de identificador da máquina, no sentido de que qualquer pedido de informação ou controlo sobre essa máquina ou para a localização dos serviços aí disponibilizados, terá de ser realizado por interacção com esse servidor.

Interface com a plataforma de base

A interface com a plataforma de base que suporta a DAMS (incluindo o sistema de operação) é obtida por uma camada que abstrai o resto do núcleo DAMS dessa plataforma em concreto. Esta é referida por `dams-low` e, no protótipo actual, esta recorre ao sistema de operação Linux (tipo UNIX), oferecendo a seguinte interface:

`int low_Exec( char *cmdline )`
: permite criar um novo processo na máquina local, sendo executado o comando indicado (utilizando as chamadas ao sistema *fork* e *execve*);

`char * low_AddHost( const char *hostName )`
: permite iniciar o sistema DAMS em máquinas remotas, recorrendo aos serviços de *remote shell* ou *secure shell*. Esta operação corresponde a lançar o programa pré-definido como sendo o representante da DAMS. Dependendo da configuração, recorre ao comandos `ssh` ou `rsh` para lançar remotamente o programa, recebendo pelo respectivo *stdout* o URI que representa o seu identificador;

`int low_ListProcs(int pIds[], int pIdsSZ )`
: permite obter a lista dos identificadores dos processos na máquina local, via a interface do sistema de operação em "`/proc`";

`void low_ProcInfo( int pid, int *ppid, int *owner, char *name )`
: permite obter informação sobre um processo na máquina local, via a interface em "`/proc`". Esta informação inclui o nome do respectivo ficheiro executável e o identificador do utilizador.

`char * low_HostInfo( void )`
: permite obter informação sobre a própria máquina onde se executa. No protótipo corrente apenas disponibiliza o nome pelo qual a máquina é conhecida no DNS.

Note-se que, outra implementação, como no caso do primeiro protótipo da DAMS sobre o PVM, esta interface pode recorrer às facilidades oferecidas por essa plataforma. Nesse primeiro protótipo utilizaram-se as primitivas `pvm_addhost()` para acrescentar máquinas à máquina virtual PVM e `pvm_spawn()` para a criação de processos em qualquer máquina nesse conjunto. Para obter informação sobre os processos em execução, utilizou-se a primitiva `pvm_tasks()`.

5.5.3 Gestão de serviços

A gestão de serviços, do ponto de vista do núcleo da DAMS, contempla o registo de servidores, sob o nome do serviço. Este registo é efectuado junto de um servidor do nú-

cleo DAMS, sendo indicado o respectivo identificador do servidor. Tipicamente, o registo é efectuado na mesma máquina, indicando a disponibilidade desse serviço nessa máquina.

Uma tabela, também gerida pelo módulo `dams-keys`, é utilizada para manter a lista de serviços disponíveis em cada nó. Esta mantém os pares <*nome de serviço*, *identificador do servidor*> para esse nó (caso a arquitectura do serviço consista em mais de um servidor, o que é aqui registado, será o primeiro interlocutor ou "ponto de entrada" do serviço, para este nó). Cada serviço, dependendo da sua implementação e arquitectura interna, regista-se nos núcleos dos nós onde se encontra disponível, ficando essa informação acessível para consulta por qualquer cliente do núcleo DAMS, até à sua remoção do núcleo, quando o serviço deixa de ser oferecido.

Na versão corrente do protótipo, cada servidor do núcleo DAMS apenas mantém a lista de serviços registados directamente, logo, não é possível por interacção com um representante do núcleo da DAMS obter os serviços que não possuam um servidor nesse mesmo nó. No caso de todas as máquinas possuírem todos os serviços que lhe podem ser requeridos, não se revela necessária a propagação desta informação, pois também existirá sempre um representante local para cada serviço.

Este registo tem de ser antecedido da própria iniciação do serviço, como uma entidade na DAMS. Isto significa que o servidor, por interacção com o núcleo da DAMS, requer o registo das suas operações que exporta, obtendo o identificador pelo qual pode ser contactado via infraestrutura de comunicação. Como esta se pretende transparente para o serviço, o identificador obtido da DAMS será opaco e depende da implementação em concreto. No caso deste protótipo, tal consiste na criação das estruturas específicas ao ORBit para que as várias operações implementadas pelo servidor do serviço, fiquem acessíveis pelos seus clientes, via o *skel* do serviço. Uma vez instanciado, o identificador atribuído pelo ORBit é utilizado como identificador do servidor, sendo com este que é efectuado o registo junto do núcleo DAMS local.

A interface oferecida é a seguinte:

`damsServ damsNewService( function table[] )`

dada uma tabela de funções que definem a implementação de um serviço (componente servidora), cria esse novo serviço. No protótipo actual, a tabela de funções tem de estar de acordo com a definição para o serviço, préviamente efectuada em IDL, sendo que

cada entrada desta tabela é vista como a implementação do respectivo *skel* no ORB. Esta operação é executada na camada de interface do serviço com a infraestrutura DAMS, sem necessidade de interacção com o servidor do núcleo.

`void damsRegService( damsHost host, char *servname, damsServ srvId )`
regista, com o nome indicado, um serviço na DAMS. Esta operação vai acrescentar à tabela de serviços no núcleo o par: *<nome de serviço, identificador do servidor>*.

`void damsDelService( damsHost host, char *name )`
remove o registo do serviço de nome indicado, sendo a respectiva entrada na tabela de serviços removida.

`serviceInfo *damsListServices(damsHost host)`
obtém a lista com os nomes de todos os serviços registados no referido nó, por listagem da tabela de serviços que existe no núcleo.

`damsServ damsGetService( damsHost host, char *servname, char *options)`
procura o serviço de nome indicado, devolvendo o respectivo identificador caso este se encontre registado.

`void damsFreeService( damsHost host, damsServ srvId )`
indica à DAMS que o processo invocador já não pretende utilizar o serviço indicado (será a operação inversa da anterior).

5.5.4 Canais de eventos

Esta abstracção é implementada no núcleo da DAMS em cada nó, onde cada canal de eventos possui uma lista de subscritores associada. Cada canal, sendo uma entidade local a um nó, pode no entanto possuir subscritores de qualquer outro nó, assim como os eventos podem ser criados, também, por qualquer outra entidade. Cada canal serve de entidade intermediária entre produtores e subscritores, possuindo a capacidade de, uma vez recebido um evento e de uma forma autónoma, notificar todos os seus subscritores.

Para a implementação dos canais de eventos, existe um módulo (`dams-evch`) que implementa uma tabela de listas. Cada lista representa os subscritores de um canal de eventos. Este suporta as operações para criar e remover cada entrada na tabela e para cada lista as-

sociada: acrescentar elemento, remover elemento e chamar uma função para cada um dos elementos.

A realização dos canais de eventos usa um identificador único para cada canal, identificador esse atribuído pela camada de comunicação (pela criação de um objecto CORBA que o representa), o que permite identificar directamente, mesmo remotamente, cada canal de eventos. Na camada de ligação ao ORBit para a implementação das operações que atendem os pedidos sobre canais de eventos, é mantida a correspondência entre esses identificadores e a respectiva entrada na tabela de subscritores antes referida. Esta será utilizada, de forma transparente, para percorrer a lista de subscritores, enviando o evento para todos eles.

A interface disponibilizada pelo núcleo DAMS é a seguinte:

`EvCh damsNewEvCh( char *key )`
: é obtido do núcleo DAMS local o identificador do canal para notificação de eventos cuja chave é a indicada. Se esta não existe, é criado um novo canal. O identificador corresponde a um novo objecto CORBA.

`void damsDelEvCh( EvCh ech )`
: é destruído o canal de eventos indicado (todos os subscritores são primeiro notificados do fim deste canal de eventos). O pedido é enviado ao próprio canal de eventos, sendo este removido da respectiva tabela no núcleo que o implementa e o respectivo objecto CORBA libertado.

`long damsSetHandler( EvCh ech, long tagmsk, Handler f )`
: o canal de eventos indicado é subscrito pelo *handler* indicado. O pedido é enviado ao respectivo canal para que o *handler* indicado seja colocado na respectiva lista de subscritores.

`void damsDelHandler( EvCh ech, Handler f )`
: o *handler* indicado é removido da lista de subscritores do canal.

`void damsPutEv( EvCh ech, damsServ sender, long tag, long datasize, void *data )`
: O evento indicado é passado ao canal referido, ficando o canal a difundir o evento por todos os seus subscritores.

`Handler damsEvCh2Handler( EvCh ech )`

obter um *handler* correspondente à operação "damsPutEv" do canal indicado, para que este possa ser tornado subscritor de outro canal.

`Handler damsFunc2Handler( void(*function)( damsServ sender, long tag, long datasize, void *data ) )`

dada a função indicada, é criado um *handler* capaz de ser usado na subscrição de um canal de eventos. Esta primitiva irá criar no ORB do invocador, um novo objecto *handler*, que fica associado à função fornecida. Este pode ser usado na subscrição de um canal de eventos.

`void damsEvPoll( bool block )`

testa e despacha os pedidos de eventos pendentes. Pode, opcionalmente, bloquear o fluxo de execução do invocador aguardando qualquer interacção do sistema DAMS. Esta primitiva destina-se às ferramentas, para que estas possam testar a existência de eventos e tratá-los.

A subscrição de um canal pressupõe a existência, no subscritor, de um mecanismo que permite a recepção dos eventos de forma assíncrona. A infraestrutura DAMS, presente em cada subscritor, disponibiliza esse mecanismo com base nas funcionalidades suportadas pela plataforma. Este mecanismo tem de ser capaz de receber o evento e invocar a função associada ao seu tratamento. Para o caso das entidades servidoras, estas já estão atentas a pedidos, sendo a entrega de um evento, tratada de forma idêntica.

Esta situação exige tratamento especial no caso das ferramentas que, sendo apenas clientes do sistema, não estão normalmente atentas à chegada de pedidos. Para que o cliente esteja disponível para essa recepção, e dependendo da plataforma, pode o mecanismo no cliente dispor de um *thread* dedicado, sempre pronto a receber estes pedidos, ou pode o programa do cliente recorrer à operação de teste, recepção e despacho destes pedidos, antes apresentada. Esta operação serve de interface para o mecanismo interno do ORBit, responsável por atender e despachar os pedidos, e assenta na função interna `giop_main_iterate(int block)`. Esta chamada, de acordo com o seu argumento, recebe e trata os pedidos pendentes ou pode ficar bloqueada a aguardar a chegada de um pedido.

O *handler* criado por `damsFunc2Handler` irá chamar a função fornecida pelo cliente, quando o canal de eventos o invocar. No caso do ORBit, isto significa criar um objecto capaz de ser invocado remotamente e em cuja implementação a função indicada será chamada.

Na operação de subscrição, o identificador deste *handler* é acrescentado à lista de subscritores do canal respectivo. A notificação pelo canal de eventos desencadeia assim a chamada (*callback*) do *handler* no subscritor com o evento como argumento. Este evento será entregue ao cliente pela invocação da função por este fornecida, onde o evento é o respectivo argumento[4].

Eventos do núcleo da DAMS

O próprio núcleo da DAMS produz eventos que podem ser subscritos pelos clientes e serviços. O protótipo reporta apenas alterações à máquina virtual DAMS, ou seja, o início ou a terminação dos processos que executam o núcleo DAMS. A API disponibilizada é a seguinte:

`EvCh damsGetBaseEvCh()` permite obter a identificação do canal de eventos usado pelo núcleo da DAMS.

O primeiro evento é gerado pelo processo DAMS ao qual foi requerida a inclusão de mais uma máquina. O evento é produzido após a confirmação de que o lançamento do processo do núcleo DAMS na máquina pretendida teve sucesso. O segundo evento será produzido, numa situação normal, pelo próprio processo DAMS que vai terminar, no canal de eventos da máquina que lhe deu origem (prevê-se que um evento equivalente possa ser produzido pelo núcleo da DAMS que dê pela sua falta). Estes eventos apresentam o seguinte formato:

```
struct {
  long evType;       // HostAdded ou HostRemoved
  long hostnameSize; // dimensao do vector seguinte
  char hostname[];   // nome da máquina
}
```

5.6 Conclusão

Os protótipos apresentados reflectem a evolução dos trabalhos, desde a primeira experiência com uma infraestrutura para monitorização sobre o PVM, e o protótipo posterior,

[4] A plataforma de suporte pode impor um limite ao tamanho da mensagem de notificação (evento) para que o seu envio possa ser atómico e não bloqueante.

segundo uma nova arquitectura interna, realizada neste caso, sobre o ORBit. Tanto o PVM como o ORBit, com auxílio do sistema de operação, permitiram o desenvolvimento de infraestruturas para monitorização adequadas aos requisitos do modelo DAMS. O segundo protótipo, pode ser adaptado a outras plataformas sem alterações significativas dos clientes, e da implementação dos serviços. Para outra plataforma será exigida a alteração na camada de *stubs* e *skels*, assim como a interface que, do lado do servidor, permite a chamada a partir dos *skels*, das operações por este disponibilizadas.

O novo protótipo permitiu evidenciar as principais diferenças entre o primeiro sistema (DAMS-1), e a segunda DAMS:

- no primeiro sistema, um processo central, o SM, com o qual todas as ferramentas têm de interagir, que possui uma arquitectura fixa para suporte dos serviços, que exige da plataforma suporte de múltiplos *threads*; no segundo protótipo, qualquer representante da DAMS pode interagir com as ferramentas e os serviços podem ser realizados em qualquer nó do sistema;

- no primeiro protótipo existem dois tipos de suporte, o SM para interagir com as ferramentas, e o LM que serve de mediador entre o SMod e o respectivo *driver*; o segundo, o núcleo DAMS é único, estando disponível em qualquer nó do sistema;

- no primeiro protótipo, os serviços têm uma arquitectura pré-definida fixa, que os organiza em SMod no processo SM, e em processos remotos em cada nó, os *drivers*; no segundo protótipo, a implementação do serviço define a arquitectura que melhor o serve, podendo-se ter serviços clientes de outros serviços como forma de estruturação de serviços mais complexos.

Note-se que o segundo protótipo permite suportar a arquitectura do primeiro, fazendo-se a correspondência entre as funcionalidades nos SMod e os serviços disponíveis aos clientes, e, quando necessário, o acesso a funcionalidades remotas, fazemos corresponder os *drivers* a serviços locais a cada nó, com os quais serviços “globais” interactuam.

Capítulo 6

Cenários de Utilização

Neste capítulo são apresentados vários casos de utilização da infraestrutura DAMS. Estes serviram de base para a validação da DAMS e dos seus serviços. Alguns casos referem-se a experiências realizadas sobre o primeiro protótipo e que serviram de motivação à concepção do presente modelo DAMS. Outros tratam-se de exemplos de utilização, ensaiados durante o desenvolvimento do segundo protótipo, ou baseados na adaptação das primeiras experiências.

6.1 Cenários estudados

No decorrer dos trabalhos realizados no contexto desta dissertação, foram várias as utilizações de ambas as versões do sistema DAMS. Estas serviram para avaliar experimentalmente as funcionalidades oferecidas pela infraestrutura DAMS. Os diversos casos estudados são aqui apresentados, de acordo com as seguintes classes de utilização, tendo em conta os principais aspectos de observação e de controlo da aplicação:

- Observação *off-line*: ilustrar sistemas para obter o traço de execução de programas. Um vocacionado para programas sobre o MPI e outro vocacionado para programas em PVM-Prolog.
- Observação *on-line*: ilustrar um sistema para obter o traço de execução de programas em Java (sobre a JVM) enquanto esta decorre;

- Controlo: ilustrar um sistema para o controlo e a inspecção, vocacionado para o suporte à depuração (*debugging*) de programas em execução;
- Observação *on-line* com controlo dinâmico: ilustrar um sistema para guiar a execução (*steering*) de uma aplicação baseada em algoritmos genéticos.

Outro ensaio apresentado tem que ver com a observação e controlo da própria arquitectura DAMS. Desenvolveu-se uma consola para gerir e observar os recursos do próprio sistema DAMS, a qual depende apenas das funcionalidades do núcleo do sistema.

Todos estes ensaios tiveram sempre por base a infraestrutura comum, podendo considerar-se que este ambiente para controlo e observação de aplicações distribuídas pode suportar todos estes casos (fig. 6.1) e outros que tenham requisitos similares.

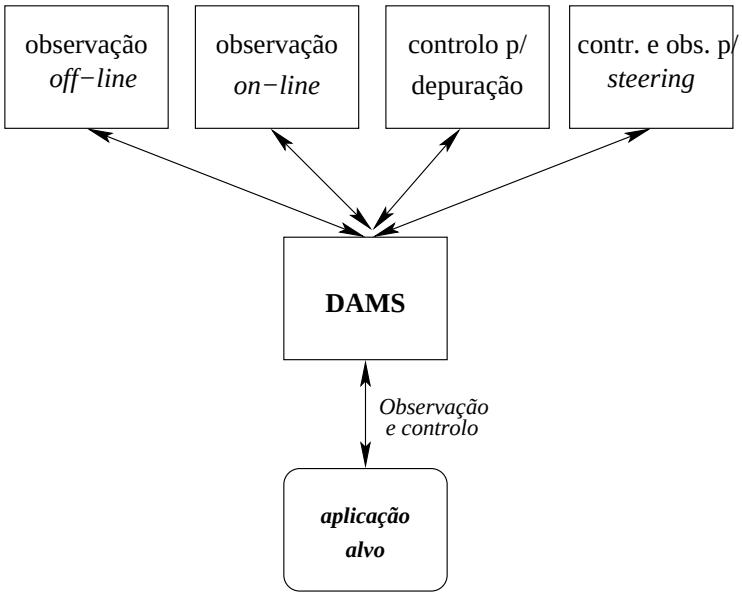

Figura 6.1: Cenários estudados

Nestes exemplos, alguns fazem uso dos serviços antes apresentados e permitem verificar a sua adaptação para diferentes objectivos, enquanto que noutros casos múltiplos serviços cooperam para obter determinada funcionalidade. Outros exemplos introduzem novos serviços sobre a infraestrutura disponível.

Alguns destes protótipos (como sejam o PDBG [69, 29]) foram primeiro desenvolvidos com base no primeiro protótipo da DAMS, e depois transpostos para o modelo corrente. Desde as primeiras experiências se podem constatar os benefícios do modelo DAMS para

o desenvolvimento de novas ferramentas [33]. Estas conclusões continuam válidas no segundo protótipo, visto este poder suportar todos os conceitos e serviços desenvolvidos com base no primeiro. Estes exemplos também mostraram a vantagem de uma infraestrutura de base comum para facilitar o desenvolvimento de novas funcionalidades e ferramentas, para reutilizar as existentes e para conseguir mais flexibilidade e robustez dos protótipos, objectivos que foram perseguidos no desenvolvimento do novo modelo e arquitectura interna da DAMS.

6.2 Consola de gestão

Uma ferramenta cliente bastante simples, baseada apenas nas funcionalidades do próprio núcleo DAMS, foi desenvolvida para servir de consola de gestão do funcionamento do protótipo do sistema. Esta consola usa apenas as funcionalidades do núcleo, permitindo que o utilizador efectue operações de consulta e gestão do estado das várias máquinas presentes no sistema, bem como dos serviços e dos processos. Esta consulta permite uma administração mínima do próprio sistema. Pode assim ser usada para lançar a execução da DAMS e configurar a sua máquina virtual, definindo quais as máquinas inicialmente sob observação, podendo poupar esse passo às restantes ferramentas.

Os comandos de utilizador suportados são os seguintes:

`add <nome>`
: acrescentar ao sistema DAMS a máquina de nome indicado (será mostrado o identificador atribuído);

`ps <nome>`
: listar todos os processos na máquina indicada;

`del <nome>`
: remover do sistema a máquina indicada;

`find <nome>`
: obter a identificação da máquina com o nome indicado (caso faça parte do sistema). Será afixada uma representação textual do identificador atribuído pela plataforma que suporta a implementação corrente da DAMS. No caso do segundo protótipo, desenvol-

vido sobre uma implementação de Corba, é mostrada a cadeia de caracteres do URI que representa aquele identificador;

`list`
: listar os nomes de todas as máquinas que constituem a máquina virtual da DAMS;

`slist <nome>`
: listar os nomes dos serviços DAMS registados na máquina indicada (portanto, suportados nessa máquina);

`quit`
: terminar a execução da consola.

Esta consola não depende de quaisquer outros serviços, podendo iniciar-se ou terminar em qualquer instante, sem alterar o sistema de monitorização ou a aplicação sob observação. Permite assim que, em qualquer instante, o utilizador efectue os comandos indicados sobre o sistema DAMS, independentemente de outras ferramentas que estejam em execução.

6.3 Observação

Nesta secção são apresentados os sistemas desenvolvidos para a observação da execução de aplicações paralelas e distribuídas, que recorrem à obtenção de um traço de execução, com vista à sua visualização ou à avaliação do desempenho da aplicação. Típicamente estes sistemas baseiam-se numa arquitectura de monitorização em que um nó central funciona como concentrador de todos os traços obtidos nos restantes nós, sendo esse o ponto de observação do sistema. Esta arquitectura pressupõe a existência de um infraestrutura capaz de transportar a informação recolhida no sistema distribuído para esse nó central, para posterior tratamento, ou então para a sua visualização durante a execução da aplicação.

6.3.1 Serviço de traço

O serviço para obter um traço, apresentado no capítulo 4, pode ser usado como base para a implementação desta classe de sistemas, nos quais são necessárias funcionalidades de observação de sequências de eventos. Ao fornecer uma infraestrutura para o transporte dos

registos dos eventos obtidos nos diversos nós, para um nó central, simplifica-se a implementação dos monitores necessários a este tipo de casos. Este serviço é neutro em relação ao formato dos registos de eventos.

Antes de descrever a utilização destes serviços nos diversos cenários ensaiados, resume-se a seguir a API disponibilizada no corrente protótipo (tendo em conta a apresentação já efectuada na secção 4.7.1) e discute-se a sua realização:

`void traceSetWorkingParam(damsServ servID, long buffsize)`
: neste protótipo, a implementação possui apenas um parâmetro para configurar o seu funcionamento, que define o tamanho do *buffer*. Este representa o número de registos que devem ser mantidos antes de os tentar passar aos respectivos consumidores e antes de começar a descartar os mais antigos. Internamente, a componente servidora de cada instância do serviço, recorre a um ficheiro como extensão do repositório mantido em memória;

`void tracePut(damsServ servID, void *b, long bsize)`
: é criado um registo com o argumento `b`, que é inserido no repositório do serviço indicado;

`void traceGet(damsServ servID, TraceBuff **b, long size)`
: o consumidor pede explicitamente ao serviço os registos mais antigos existentes, até ao número máximo indicado por `size`. No vector `b` ficará esse número de apontadores para registos, ou menos (`TraceBuff` é uma estrutura que contém o tamanho do registo seguido desse mesmo registo). Os registos obtidos são retirados do servidor;

`void traceSetProd(damsServ servID, damsServ prod)`
: o produtor regista junto do serviço a sua função de produção. Quando requerido pelos consumidores (quando de um `traceGet` ou `traceFlush`) o serviço invoca a função registada para que esta lhe entregue os registos (nesta situação, repete o pedido a todos os produtores registados) e depois passa a informação recolhida ao consumidor que a pediu ou a todos os consumidores registados, consoante a operação que lhe deu origem;

`void traceSetCons(damsServ servID, damsServ cons)`
: o consumidor regista no serviço indicado uma sua função, responsável por receber os

registos. Quando necessário, pelos parâmetros antes apresentados, o serviço invoca esta função, entregando-lhe os registos existentes (nesta situação esta operação repete-se com cada registo para todos os consumidores).

`void traceFlush(damsServ servID)`
: pede todo o traço existente, sendo este entregue, pelo serviço, utilizando os registos de consumidores, efectuados com a operação anterior.

Em cada máquina deve estar disponível uma instância deste serviço, responsável por coligir os registos de eventos detectados nessa máquina. Uma arquitectura de monitorização que pretenda reunir todos os registos dos eventos num nó central do sistema, tem de requisitar cada um destes serviços e registar o serviço no nó central, como consumidor de todos os restantes (arquitecturas mais complexas são possíveis), para que este funcionem como concentrador na raiz desta arquitectura. É neste último que a ferramenta recolhe toda a informação obtida. De acordo com o objectivo pretendido, os registos podem ser passados para a raiz, com maior ou menor latência. Nos casos extremos, devem ser passados de imediato (aproximando-se de uma observação *on-line*), ou então, só serem recolhidos por pedido expresso da ferramenta (por exemplo, efectuando `traceFlush` após a terminação da aplicação).

Descrição da implementação do serviço de traço

Tal como para o núcleo DAMS, as funcionalidades deste serviço são oferecidas através de uma interface, recorrendo esta em muitas das operações a um servidor que permite a partilha do repositório de registos no serviço, entre os vários clientes e serve de intermediário entre esses clientes e os geradores desses registos (ver fig. 6.2). Em cada nó onde o serviço se encontra disponível, existe um destes servidores.

Na organização deste serviço adoptou-se o mesmo tipo de arquitectura *software* com múltiplos níveis, como visto para o núcleo DAMS. O servidor que suporta o serviço baseia-se num módulo interno, chamado `trace-low`, que inclui suporte às seguintes operações:

- Para a implementação de um fila de registos, que podem ser inseridos e removidos segundo a uma disciplina FIFO (*first in first out*):

  `trace_put, trace_get, trace_tail, trace_head`

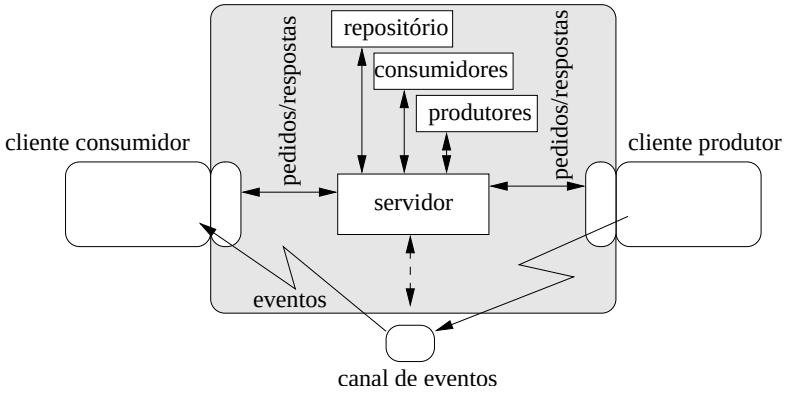

Figura 6.2: Implementação do serviço de traço

- Cada fila tem também associada duas listas de identificadores (que são vistos como um tipo opaco), com operações para a sua inserção e remoção, e ainda para percorrer todos os seus elementos:

  ```
  trace_addprod, trace_addcons, trace_delprod, trace_delcons,
  trace_iterateprod, trace_iteratecons
  ```

  Estas listas implementam, respectivamente, os conjuntos dos consumidores e dos produtores dos registos.

Estas operações correspondem à verdadeira implementação do servidor do serviço de traço, sendo estas disponibilizadas, no segundo protótipo, numa interface oferecida via Corba. Este módulo prevê a existência de mais de uma instância do serviço no mesmo espaço de endereçamento. Inclui assim a possibilidade de gerir mais de uma fila de registos, mas tal não foi explorado na implementação corrente do serviço de traço.

Internamente a este módulo, está definido um limite ao tamanho do repositório em memória, dependente da implementação, a partir do qual os registos passam a ser guardados no sistema de ficheiros local. O objectivo é poupar o uso de espaço de memória e permitir a recolha de grandes traços, como nas situações em que se pretendem os traços apenas após a terminação da aplicação alvo.

As várias operações suportadas pelo serviço, tornam-se acessíveis à respectiva API nos clientes ao serem registadas, como suporte ao serviço junto do núcleo DAMS. Aí, terá de existir um suporte específico da camada que serve de interface com a plataforma de comunicação utilizada. Neste caso, a implementação recorre à camada de interface com o ORBit, na

qual cada função passa a poder ser remotamente chamada por meio de um *skel*. Esta camada de interface esconde a existência da camada Corba às primitivas internas do servidor.

O mecanismo para passar os registos aos subscritores consumidores terá de possuir também um suporte específica da camada de comunicação utilizada. A operação para envio dos registos aos interessados tira partido do mesmo mecanismo para os canais de eventos apresentado anteriormente, mas sendo agora a operação a chamar no cliente a indicada por este na subscrição.

Nesta secção, descrevem-se as experiências efectuadas com vista à obtenção de traços de execução em diversos cenários de aplicações paralelas e distribuídas. São apresentados os seguintes casos:

1. obter um traço de execução das chamadas à interface PVM por parte de um programa Prolog, no sistema PVM-Prolog [93], após o terminação da execução;
2. instrumentação da biblioteca MPI [101, 102], com vista à obtenção, após a aplicação terminar, de um traço de execução com o registo das comunicações efectuadas;
3. obter durante a execução (*on-line*) um traço da execução de aplicações em Java [120], com múltiplos processos distribuídos.

Nos dois primeiros casos, como só após a execução completa da aplicação, é que se pretende obter o respectivo traço, a monitorização passará pelas três fases seguintes:

1. configuração da infraestrutura de monitorização;
2. recolha da informação junto de cada processo;
3. transporte e fusão, num único nó, de toda a informação recolhida.

No terceiro caso, queremos disponibilizar informação, ao cliente, durante a execução da aplicação. Temos assim uma situação onde a recolha da informação, e o seu envio à ferramenta, ocorre em simultâneo com a execução da aplicação. Tem, por isso, de se procurar minimizar a latência no transporte da informação recolhida para o nó central, ao mesmo tempo que se procura minimizar a perturbação introduzida na aplicação pela monitorização.

Para tal, a informação recolhida tem de ser mínima, em termos do número de eventos e do volume da informação coligido.

Todos estes casos assentam no serviço genérico de traço antes apresentado. As arquitecturas destes casos seguem o esquema apresentado na figura 4.13, adaptando em cada caso a instrumentação efectuada e a ferramenta que recolhe o traço, que são específicas a cada caso.

6.3.2 Monitorização de aplicações PVM-Prolog

O PVM-Prolog [93, 32] é uma extensão à linguagem Prolog para permitir a interacção entre vários processos Prolog usando as primitivas do sistema PVM. Esta extensão permite a passagem de termos Prolog entre processos e a avaliação de golos Prolog em máquinas remotas. A interface PVM pode também ser utilizada para a interacção dos processos executando programas em Prolog com outros processos executando programas desenvolvidos em outras linguagens, nomeadamente a linguagem C.

Sentindo-se a necessidade de monitorizar estas aplicações, em particular as interacções entre os vários processos, com vista a avaliar a correcção e o comportamento dos programas, ou a suportar a sua depuração, foi desenvolvido um monitor, sobre a primeira versão da DAMS. O sistema implementado é adaptável (com vantagens) à nova versão do modelo DAMS, mantendo-se assim os benefícios na altura reconhecidos [24], sendo essa adaptação discutida no fim desta secção.

Este caso ilustra um exemplo em que o modelo de programação se distancia do modelo imperativo, habitualmente usado no desenvolvimento em aplicações paralelas. Este modelo assenta, no entanto, na plataforma PVM, sendo ao nível da sua interface com a linguagem Prolog que se identificou a necessidade de efectuar a instrumentação. Conseguem-se assim registar as interacções via PVM, sendo no entanto possível obter informação sobre os termos Prolog envolvidos em cada interacção. Com uma ferramenta genérica para o PVM, a informação recolhida descreveria os tipos de dados usados no protocolo estabelecido sobre o PVM, e não os usados ao nível do programa Prolog.

Arquitectura do monitor

No primeiro protótipo da DAMS foi desenvolvido um serviço de monitorização, com o objectivo de recolher registos de eventos em cada máquina do sistema DAMS. O serviço divide-se numa parte central com a qual a ferramenta dialoga e outra remota, da qual uma instância se executa em cada máquina. A ferramenta consiste numa consola, com a qual o utilizador interactua para dar comandos, que permitem iniciar o monitor, verificar o seu estado e pedir a recolha do traço de execução de cada nó.

A componente remota do serviço disponibiliza um canal FIFO[1] em cada máquina, pelo qual são entregues ao serviço os registos dos eventos detectados no PVM-Prolog. Utiliza-se, para essa detecção, um conjunto de primitivas de uma biblioteca de interface, disponibilizada para esse fim (descrita a seguir) que permite instrumentar a interface do Prolog com o PVM. Todos os registos vão sendo guardados localmente em cada nó, aguardando-se que o componente central os requeira.

A configuração da infraestrutura descrita é realizada pela ferramenta, por interacção com a DAMS, requerendo o serviço de monitorização e, depois, requerendo a este a sua activação em cada nó da máquina virtual do PVM. Antes do início da execução da aplicação, há que configurar a arquitectura da máquina virtual, ou seja, registar todas as máquinas participantes no ambiente sob monitorização da DAMS.

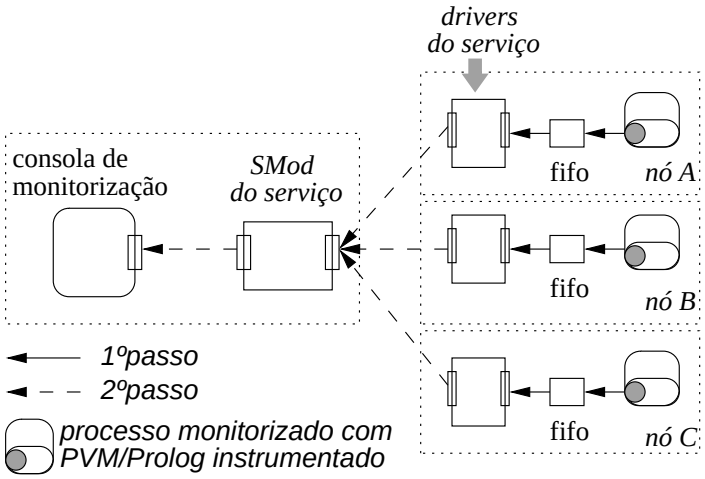

Figura 6.3: Monitorização do PVM-Prolog

[1]Implementado no protótipo, usando um *named pipe* do sistema Unix.

Cada máquina sob observação (ou nó) vai executar uma instância do *Local Manager* (LM) onde o respectivo *driver* de monitorização se encontra sob o controlo do *Service Module* (SMod) de monitorização que executa sobre o *Service Manager* (SM) (ver figura 6.3). Cada um destes *drivers* cria o seu FIFO, ficando disponível para receber os registos que lhe enviem por esse canal. Como se trata de uma abordagem para recolha de informação a ser consumida *a post-mortem*, basta fazer fluir essa informação após a terminação da execução da aplicação. Assim, num primeiro passo, os registos são guardados localmente, na sua totalidade, evitando concorrer no processamento e na utilização da largura de banda da rede, evitando assim interferir com a aplicação.

A ferramenta, encontra-se implementada utilizando a interface oferecida pelo serviço desenvolvido, que inclui as seguintes principais operações:

PrfInit() — inicia a ligação ao serviço de monitorização e regista o processo corrente (invocador) como consumidor das notificações do serviço;

PrfSetup() — inicia os respectivos *drivers* em todos os nós no sistema DAMS (cada nó passa a dispor do seu canal FIFO ligado ao respectivo *driver* pronto a receber registos);

PrfCollect() — pede ao serviço o traço obtido da *task* com o identificador indicado. Para tal o *driver* responsável pelo nó será contactado para que este envie o referido traço. Este traço é gravado pelo SMod, no ficheiro cujo nome também é indicado;

PrfCollectAll() — pede o traço de todos os *drivers*, gravando o resultado no ficheiro de nome indicado;

PrfGetNotification() — testa se existe uma notificação pendente. Se existir, recebe uma estrutura que descreve um dos seguintes eventos: início de um processo da aplicação ou o fim de um processo;

PrfQuit() — termina a ligação ao serviço.

Depois de iniciar a sua ligação à DAMS, o serviço é requerido e iniciado utilizando as duas primeiras operações. A consola entra então num ciclo de aceitação/execução dos comandos dados pelo utilizador e de teste da chegada de notificações. De cada vez que uma notificação é detectada, a consola afixa uma mensagem descrevendo o acontecimento. Estes eventos são detectados e produzidos pelo *driver*, ao receber os registos que descrevem o início ou o fim dos processos instrumentados.

Os comandos disponibilizados ao utilizador permitem: listar os processos que estão ligados ao serviço (com base nos eventos recebidos); sair da ferramenta, o que implica o término do serviço e terminação da recolha do traço. Quando o utilizador deseja, tipicamente depois da terminação de todos os processos, pode dar um comando para coligir num ficheiro os registos efectuados por um determinado *driver*, ou por todos. Tal desencadeia, no SMod, a recolha dos registos armazenados nos vários *drivers* remotos. Este corresponde ao segundo passo (fig. 6.3), sendo todo o traço guardado num ficheiro, para posterior análise. A informação recolhida pode, em seguida, ser reordenada e guardada no formato pretendido.

A visualização do traço obtido foi conseguida por um pós-processamento que ordena globalmente os registos pelas suas estampilhas data-hora e os converte para o formato NPICL[129]. Para tal, fez-se corresponder cada processo (*task*) PVM-Prolog a um nó no modelo do NPICL; cada envio de predicados para avaliação remota corresponde a um envio de mensagem no NPICL. Foi assim tornado possível analisar a execução das aplicações, com qualquer ferramenta compatível com esse formato. No caso presente, o Paragraph[55] foi a ferramenta usada para visualizar e analisar o comportamento da execução.

Instrumentação do PVM-Prolog

A instrumentação foi efectuada ao nível da interface da máquina abstracta do Prolog com a biblioteca PVM. Esta interface compreende a definição de predicados Prolog, nos quais são efectuadas as chamadas à API do PVM. Nesta interface, antes de efectuar a respectiva chamada ao PVM, os termos Prolog a serem passados aos processos remotos, têm de ser convertidos a partir da representação usada no interpretador, para os tipos de dados reconhecidos pelo PVM (inteiros, cadeias de caracteres, etc.), segundo um protocolo estabelecido no PVM-Prolog. Operação inversa é efectuada com os resultados obtidos, que têm de ser convertidos em tipos da máquina abstracta Prolog, quando recebidos, para permitir a respectiva unificação.

Exemplo: o envio de um termo Prolog a um processo remoto significa que este é codificado numa mensagem, incluindo a representação da "tag" que indica o tipo de termo e, dependendo desta, a aridade das listas e respectivos elementos, os átomos com indicação dos respectivos tipos e valores, etc. Na recepção, há que reconverter a mensagem em termos Prolog e só depois passá-los ao interpretador, executado no contexto do processo destinatário.

É ao nível desta interface que a instrumentação foi efectuada, podendo a informação recolhida ter assim relação com o nível de abstracção do Prolog e não apenas com o nível dos tipos de dados do PVM, de pouco significado para o programador Prolog.

A instrumentação pode assim registar a informação que diz respeito apenas às chamadas à interface PVM-Prolog, sem descrever os detalhes internos que dizem respeito à conversão e empacotamento em mensagens. Também a informação registada indica a informação trocada entre predicatos, em termos da respectiva aridade enviada ou recebida e não do número de *bytes*. Esta informação torna-se assim mais útil para o programador Prolog, a quando da visualização do traço de execução da aplicação e para a tentativa de detecção de erros.

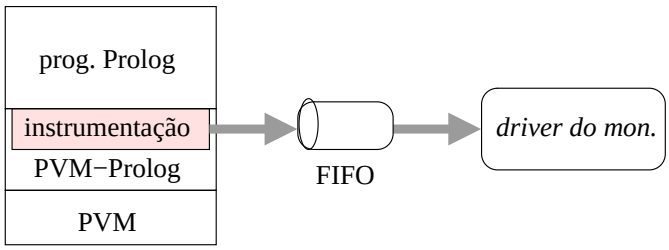

Figura 6.4: Instrumentação no PVM-Prolog

Uma versão instrumentada do PVM-Prolog foi realizada por reescrita dos predicados da interface do sistema, por forma a permitir registar os eventos de chamada e de término de cada um desses predicados (fig. 6.4). A descrição de cada um destes eventos inclui a seguinte informação:

- identificação da origem do evento (*task id*);
- data-hora da ocorrência, obtida pela chamada `gettimeofday()`;
- qual o predicado PVM-Prolog chamado;
- se está a invocar ou a retornar do predicado;
- se está a invocar, descreve os argumentos e, dependendo da primitiva, pode incluir a identificação de interlocutores, o tamanho que os argumentos ocupam na mensagem, etc;
- se está a retornar, indica o resultado da chamada ao PVM.

Esta instrumentação encontra-se implementada com base nas seguintes funções integradas numa biblioteca de instrumentação construída para interactuar via o canal FIFO (ver fig. 6.4) com o serviço de monitorização:

`ProfileInit( char *id )`
: estabelece a ligação ao canal FIFO local. A cadeia '`id`' identifica este processo na plataforma PVM, sendo este identificador associado a todos os registos recolhidos. É enviado um primeiro registo para o serviço de monitorização, indicando a existência de um novo processo e este mesmo identificador;

`ProfileTrace( char *reg )`
: cria um registo, sendo a cadeia indicada a descrição de um evento. Esta função é também responsável por incluir no registo o identificador antes indicado em `ProfileInit` e a data-hora, enviando esta informação ao serviço pelo canal FIFO;

`ProfileFinish()`
: é enviado um registo indicando a terminação deste processo via o respectivo canal, seguindo-se o fecho desse canal.

Ao próprio programador é facultada a possibilidade da introdução de registos específicos no traço, usando um novo predicado criado para o efeito: `instrument(S)`. Este serve de interface com a função `ProfileTrace`, sendo `S` o argumento passado a essa função. O programador pode assim inserir explicitamente novos registos no traço a partir de qualquer ponto no seu programa Prolog. Por exemplo:

```
instrument('Inicio-calc'), calc(X),instrument('Fim-calc')
```

resultará num traço onde todos os registos obtidos na avaliação de `calc(X)` estarão delimitados por registos contendo na sua descrição `Inicio-calc` e `Fim-calc`.

Conclusão

Dada a neutralidade da abordagem seguida, relativamente à informação registada, a DAMS e o serviço aqui apresentado podem facilmente ser utilizados em outros ambientes com necessidades de monitorização semelhantes. A comprovação desta flexibilidade foi

ilustrada, ao desenvolver-se um sistema, com base neste, para obter o traço de execução de aplicações sobre o MPI, que se apresenta na secção seguinte.

Como pretendido, a dependência da ferramenta, é apenas relativa à sua interface com a API oferecida pelo serviço. De forma idêntica, na aplicação, a instrumentação baseia-se apenas na biblioteca apresentada para interagir com o serviço de monitorização.

O ambiente descrito pode facilmente ser adaptado ao novo modelo de DAMS, através da reescrita do serviço de monitorização. Neste caso existirá um serviço central que desempenha as funções do SMod no modelo antigo, e cada um dos nós disponibiliza uma instância de um serviço de desempenha as funções do *driver* no modelo antigo. Mantendo-se as interfaces utilizadas pela ferramenta e a oferecida pela biblioteca para instrumentação, quer a ferramenta quer a instrumentação efectuada no sistema PVM-Prolog, continuam inalteradas.

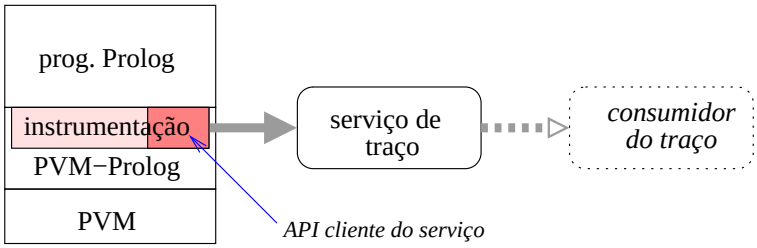

Figura 6.5: Instrumentação de PVM-Prolog na nova DAMS

No entanto, veremos que no caso do MPI, para a implementação das APIs necessárias sobre a nova infraestrutura, se pode tirar partido das facilidades já disponibilizadas pelo serviço de traço do novo protótipo, tornando mais fácil a sua implementação. Assim, e no que diz respeito à instrumentação efectuada no PVM-Prolog, esta mantém-se, sendo que agora a implementação da biblioteca para instrumentação pode utilizar a API de cliente oferecida pelo serviço de traço para enviar os registos recolhidos (fig. 6.5).

```
ProfileTrace(reg)
{
  tracePut(s, buildRecord(reg))
}
```

As notificações de início e fim de cada processo são suportadas pelos canais de eventos associados aos serviços de traço, sendo gerados eventos pela biblioteca de instrumentação a quando do `ProfileInit` e do `ProfileFinish`.

```
ProfileInit()
{
  damsInit()
  s = damsGetService("Trace")
  ec = traceGetEvCh(s)
  damsPutEv(ec, buildInitEv())
}
```

```
ProfileFinish()
{
  damsPutEv(ec, buildFiniEv())
  damsFreeService(s)
  damsEnd()
}
```

Do lado da ferramenta, a interface necessária é suportada sobre as interfaces do serviço de traço e do núcleo da DAMS, sendo que:

- existe um nó central onde o serviço de traçado é utilizado como concentrador dos restantes, sendo com este que a ferramenta interactua para receber o traço;
- a recolha central do traço é efectuada após o término da aplicação, tal como anteriormente, mas agora recorrendo às instâncias do serviço de traço, para fazer chegar os registos ao nó central.

As primitivas de inicialização realizam a activação da infraestrutura, para obter a mesma funcionalidade, antes obtida pelo serviço específico:

```
PrfInit()
{
  damsInit()
  s = damsGetService("Trace")
  traceSetWorkingParam(s,ALL)
  ec = traceGetEvCh(s)
  damsSetHandler(ec,
     clientHandler())
}
```

```
PrfSetup()
{
  traceSetCons(s, clientRec() )
  foreach host in damsListHosts()
    t=damsGetServive(host,"Trace")
    traceSetWorkingParam(t,ALL)
    if t <> s then traceSetCons(t,s)
}
```

em que *clientHandler()* é responsável por receber o evento e torná-lo disponível à primitiva PrfGetNotification; e *clientRec()* é responsável por receber o traço e escrevê-lo num ficheiro indicado por PrfCollectAll:

```
PrfCollectAll( filename )
{
   f = openFile( filename )
   traceFlush(s)
}
```

```
PrfGetNotification()
{
   if damsEvPoll( NoBlock )
      then processNotification()
}
```

```
PrfQuit()
{
  damsDelHandler(ev, clientHandler())
  damsFreeService(s)
  damsEnd()
}
```

A interface do antigo serviço é assim fácilmente obtida apenas como uma nova API sobre

o serviço de traço já existente na DAMS. Algumas vantagens relativamente ao primeiro protótipo, obtidas com a nova concepção do modelo da DAMS apresentado nesta tese:

- a infraestrutura de comunicação é a já existente, não sendo por isso necessário implementar novos serviços para obter o traço, nem estabelecer uma comunicação sobre um canal FIFO, implementada especialmente para este efeito;
- a infraestrutura pode ser configurada para também ser usada para operar num modo *on-line*, sem necessitar de grandes alterações;
- uma ferramenta que pretenda interactuar apenas com um processo de uma dada máquina (ou apenas um subconjunto), pode usar directamente os respectivos serviços de traço, dispensando os intermediários.

6.3.3 Monitorização de aplicações MPI

Descreve-se a seguir o desenvolvimento de um sistema de monitorização de uma aplicação sobre o sistema MPI [101, 102], equivalente aos habitualmente usados para obter o traço de execução.

Neste caso o objectivo era recolher informação sobre as chamadas da aplicação ao nível do modelo MPI. Pretendia-se *a posteriori*, verificar as interacções ocorridas entre os vários processos da aplicação e assim suportar a avaliação do seu desempenho e/ou a sua correcção. Pretendia-se manter a compatibilidade desta fase com as seguintes fases de análise e visualização, ou seja, manter a compatibilidade com outras ferramentas já existentes e desenvolvidas separadamente por outros grupos. Neste caso, tratando-se da implementação MPICH[52, 51], pretendeu-se manter a compatibilidade com os visualizadores da família Jumpshot[17].

Arquitectura do monitor

A arquitectura típica para este tipo de monitorização é idêntica à realizada para o PVM-Prolog. Este é o caso dos monitores que fazem parte do ambiente MPE (*Multi-Processing Environment*) que acompanha o MPICH. A abordagem seguida foi assim idêntica à usado

para o PVM-Prolog. As primitivas para instrumentar a biblioteca MPI são também equivalentes. Este protótipo foi logo desenvolvido sobre o novo modelo, seguiu-se por isso a abordagem apresentada no fim da secção anterior.

Na fase inicial há que configurar a arquitectura da máquina virtual DAMS, ou seja, registar todas as máquinas participantes do ambiente MPI como nós sob observação da DAMS. É necessário que os serviços de traço sejam encadeados entre si de modo a permitir à ferramenta recolher junto de um serviço que fica como central, o traço com toda a informação recolhida. A activação da configuração especificada é conseguida, pela ferramenta desenvolvida, por interacção com o núcleo DAMS e com os serviços de traço de cada nó.

Como se trata de uma abordagem para recolha de informação a ser consumida *post-mortem*, basta fazer fluir essa informação após a terminação da execução da aplicação. Deve-se assim configurar cada instância do serviço de traço, para armazenar a totalidade dos registos localmente.

A ferramenta funciona como o monitor central, tirando partido do suporte dado pela infraestrutura DAMS. A recolha dos vários traços é efectuada por esta ferramenta, num segundo passo (tal como para o PVM-Prolog), quando a execução termina — os eventos recebidos pelo canal associado ao serviço permitem à ferramenta detectar o fim da aplicação. A informação recolhida deve ser reordenada e convertida no formato pretendido. Para manter a compatibilidade com o Jumpshot, os registos recolhidos devem conter informação idêntica à obtida pelos monitores para o Jumpshot, e o ficheiro final deve ficar num formato compatível com as ferramentas que acompanham o MPICH.

Instrumentação da biblioteca

Para detectar as chamadas à biblioteca MPI e registar assim a informação relevante para esses eventos, recorreu-se à introdução de código de instrumentação na API da referida biblioteca. A norma MPI contempla já uma interface (PMPI) à qual se recorre para obter bibliotecas instrumentadas. No caso da implementação MPICH, esta oferece, junto com o seu sistema MPE (*Multi-Processing Environment*), ferramentas para que as mais variadas formas de instrumentação possam ser mais facilmente efectuadas. Estas permitem definir um conjunto de regras que descrevem as funções da interface MPI a serem alteradas e qual o código a introduzir, antes e depois da verdadeira chamada à biblioteca MPI. Um pré-

processador (`wrappergen`) baseia-se nesta descrição para gerar o código para a interface PMPI, obtendo-se uma biblioteca instrumentada.

Neste caso em particular, assume-se que, quando a aplicação se inicia, esta já pode estabelecer a ligação ao serviço local de traço na DAMS (caso este esteja activo). Tal é conseguido através da instrumentação efectuada à primitiva `MPI_Init`, já que esta primitiva marca o arranque de cada processo como membro da aplicação sobre o MPI. A primitiva `MPI_Finalize` foi instrumentada para indicar a finalização do traço nesse processo por um evento, dando ao monitor central a possibilidade de, quando entender, pedir a todos os serviços de traço a informação recolhida por cada um.

No sistema MPE as bibliotecas instrumentadas recorrem tipicamente às primitivas `MPE_Log_event()`, para registar os eventos relevantes e ainda às primitivas `MPE_Log_send()` e `MPE_Log_receive()` para registar os casos de envio e recepção de mensagens. Existem ainda duas funções, `MPE_Init_log()` e `MPE_Finish_log()`, que permitem, por instrumentação das primitivas `MPI_Init()` e `MPI_Finalize()`, respectivamente, iniciar e terminar o registo do traço da execução.

De forma semelhante à do caso do PVM-Prolog, mas para ser adequada à nova DAMS, foi implementada uma primitiva que permite enviar para o serviço local de traço a informação recolhida, `ProfileTrace()`, e mais duas que iniciam e terminam o processo: `ProfileBegin()` e `ProfileEnd()`. Estas primitivas seguem o desenho antes apresentado para o caso do PVM-Prolog, cumprindo os mesmos propósitos. No entanto, neste caso, estas foram adaptadas ao sistema MPI. Para tal, a identificação do originador (identificação do processo) e a data-hora de ocorrência do evento passam a ser obtidos via interface MPI. De notar que o MPI oferece a função `MPI_Wtime()` para obter, com grande resolução[2], o tempo decorrido desde um instante de referência, podendo ainda a implementação suportar a sincronização dos relógios de todas as máquinas envolvidas.

A instrumentação efectuada à função `MPI_Init()` permite estabelecer a ligação ao serviço de traçado e indicar, via um evento, o início do processo. No caso do `MPI_Finalize()`, é gerado o evento indicando o fim do processo, e terminada a ligação ao serviço de traçado. Esta instrumentação é equivalente ao seguinte código:

[2]Segundo a norma, a resolução deve ser a maior permitida pela plataforma subjacente à implementação do MPI.

```
MPI_Init(...)
{
  ret = PMPI_Init(...)
  PMPI_Comm_rank(MPI_COMM_WORLD, &myid)
  ProfileBegin( myid )
  return ret
}
```

```
MPI_Finalize()
{
  ProfileEnd()
  return PMPI_Finalize()
}
```

Na fase de recolha de informação sobre as chamadas ao MPI que sejam de interesse e seu envio para o serviço de traçado, recolhe-se a informação que permita descrever os eventos de forma equivalente aos sistemas existentes no MPE. Esta é obtida por um evento de entrada na função e outro no retorno da mesma. Como exemplo apresenta-se o pseudo-código da instrumentação realizada nas primitivas `MPI_Send()` e `MPI_Recv()`:

```
MPI_Send(...)
{
  ProfileTrace( buildRec(MPI_Send_B,
      dest, tag, size))
  ret = PMPI_Send(...)
  ProfileTrace(buildRec(MPI_Send_E, ret))
  return ret
}
```

```
MPI_Recv(...)
{
  ProfileTrace( buildRec(MPI_Recv_B))
  ret = PMPI_Recv(...)
  ProfileTrace( buildRec(MPI_Recv_E,
      src, tag, size))
  return ret
}
```

A informação registada inclui a identificação do processo e do instante em que ocorre e, nos casos necessários, a identificação do interlocutor e volume de dados trocados.

Para que a ferramenta, ou seja, o monitor central, determine o fim da aplicação, cada canal de eventos associado aos serviços de traço é usado para indicar o início e o fim de cada processo. O monitor central mantém um contador do número de processos activos, sendo actualizado de acordo com os eventos recebidos. Assume-se que, após o início do primeiro processo, a aplicação termina quando este contador voltar a zero. Nessa altura, os traços são requeridos pela chamada da primitiva `traceFlush()` e toda a informação recebida é gravada em ficheiro.

Segue-se, em fase posterior e se necessária, a ordenação dos eventos recebidos para que se possa obter um único traço de execução, consistente para toda a aplicação. É nesta fase também que a ordenação pode ter em conta as correcções necessárias para que a ordem causal não seja violada, ou que se podem aplicar correcções que tentem compensar a interferência da monitorização, assim como a conversão do traço para o formato pretendido pelas ferramentas que o analisam. Neste caso optou-se por converter o traço final no formato CLog[17], para compatibilidade com as ferramentas de análise que acompanham o MPICH/MPE.

Concluindo

Demonstra-se a facilidade de inclusão, no ambiente DAMS, de um mecanismo para monitorização de aplicações em MPI, apenas com o desenvolvimento de uma simples camada de instrumentação, capaz de recolher a informação pretendida e enviá-la para um serviço já existente, no caso, o de traço. Este, junto com a restante infraestrutura da DAMS, suportam o armazenamento e transporte para o nó central, onde o traço da aplicação é obtido como pretendido. O mecanismo de eventos oferecido permite que a ferramenta central tenha informação sobre o estado global de execução dos vários componentes da aplicação.

Com pouco esforço podemos tirar partido desta infraestrutura para a adaptar a novos objectivos e novas ferramentas. Por exemplo, para obter um monitor *on-line* teremos de efectuar as seguintes alterações: garantir que a instrumentação regista eventos separados para a entrada e saída de cada função do MPI;[3] os serviços de traçado passam essa informação com o mínimo de atraso, dependendo da configuração, até chegar ao monitor, em vez de a armazenarem e ficarem a aguardar que esta lhes seja pedida. Dado o impacto que este tráfego de informação pode trazer à própria aplicação, convém que a instrumentação seja mínima.

Outro caso será dispor deste monitor em simultâneo com outros serviços, por exemplo, o de depuração visto na secção 6.4. Esta integração pode trazer funcionalidades acrescidas ao permitir visões concorrentes, oferecidas por diversas ferramentas. Permite a visualização da execução efectuada, sobre a forma do traço de execução, ao mesmo tempo que é efectuada a depuração simbólica do código. A quando de *breakpoints*, pode-se mais facilmente verificar o comportamento da aplicação durante a execução até chegar a esse ponto, e assim, melhor identificar as possíveis causas de problemas —serve de complemento à visão do estado da pilha de execução de um processo, permitindo uma visão do "caminho" que toda a aplicação seguiu até chegar ao estado que se está a analisar.

Este desenho de monitor, além de semelhante ao visto para o PVM-Prolog, pode ser adaptado a outras situações envolvendo outras plataformas de suporte às aplicações. As alterações a efectuar estão confinadas à instrumentação das respectivas bibliotecas e adaptação das primitivas de instrumentação para a informação relevante nesta plataforma.

[3] Este é já o caso do pseudo-código anterior, mas tal não se passa no MPE.

6.3.4 Monitorização de aplicações Java

Na sequência de um trabalho de fim de curso da LEI [59], em cuja orientação o autor desta tese esteve envolvido, foi desenvolvido um sistema para recolha de traços de execução de programas Java na JVM [120]. Este sistema inclui um componente para instrumentação da aplicação, um monitor que colige informação sobre as alterações de estado da máquina virtual Java e uma "ferramenta" que recolhe a informação do monitor e a pode afixar no ecrã para visualização pelo utilizador, ou gravar em ficheiro o traço dessa execução (fig. 6.6).

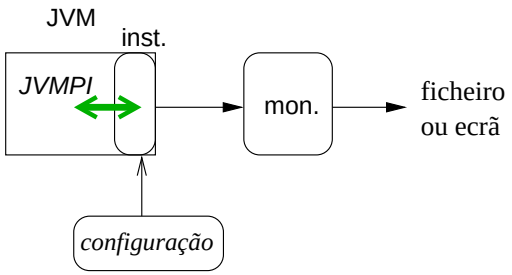

Figura 6.6: Monitorização de uma máquina JVM

Apresenta-se aqui a técnica usada para instrumentar a JVM e descreve-se como esta arquitectura foi depois integrada na DAMS para a monitorização de várias JVM distribuídas numa rede local, com vista a suportar a monitorização de aplicações Java distribuídas.

Instrumentação usando JVMPI

Para a instrumentação da máquina virtual e recolha de informação, o sistema recorre à interface JVMPI (*Java Virtual Machine Profiling Interface*), suportada pela JVM da Sun. Esta permite detectar vários tipos de eventos ao nível da máquina JVM, como o carregamento de bibliotecas, o início e fim de *threads* ou a chamada de métodos. Permite também obter, da JVM, informação complementar que permite identificar e descrever os eventos detectados. Por exemplo, quando é detectada a chamada de um método, pode então ser obtido o identificador (nome) do método e respectiva classe e a identificação do *thread* que executa o método. Para obter estes eventos, uma biblioteca dinâmica para instrumentação foi desenvolvida pelo aluno utilizando essa interface, sendo esta ligada à JVM a quando do seu arranque, sempre que se pretenda utilizar esta instrumentação.

Dada a enorme quantidade de eventos possíveis durante a execução de um programa,

alguns internos às classes do sistema Java e portanto normalmente desconhecidos do programador, a instrumentação desenvolvida neste trabalho recorre a uma configuração definida pelo utilizador, através da qual se indicam os tipos de eventos que interessa registar. Torna-se possível indicar quais os métodos relevantes, sendo os restantes ignorados pela instrumentação. É assim possível obter o traço de execução de um programa Java onde, por exemplo, fica registada a evolução dos diversos *threads*, e as chamadas de métodos que considerarmos relevantes, para a observação da aplicação, ignorando todos os restantes, como sejam os de inicializações da própria JVM e do carregamento de classes *standard*.

Aquela configuração é especificada num ficheiro de texto, no qual são indicadas as classes e respectivos métodos a registar. O utilizador pode editar directamente este ficheiro, ou utilizar uma ferramenta gráfica para a sua edição, o que lhe dá a possibilidade de activar ou desactivar o registo das classes e métodos conhecidos.

A interface JVMPI não permite aceder aos argumentos com que os métodos são chamados. Esta limitação torna impossível saber, a este nível, qual o destinatário de qualquer interacção entre *threads*, mesmo sabendo quais os métodos relevantes. No entanto, foi desenvolvida uma interface que permite ao próprio programador anotar o seu código com os seus próprios eventos, podendo assinalar assim também as interacções.

Estes eventos, de nível utilizador, são gerados por chamada, no programa Java, do seguinte método:

```
public native void
  User_Event_Markup(int type, String msg, int mark1, int mark2)
```

Neste, os argumentos permitem indicar:

`type` — o identificador do evento, à escolha do programador;
`msg` — uma cadeia de caracteres que descreve o evento;
`mark1`, `mark2` — dois identificadores extra.

No caso das interacções, os dois últimos argumentos são usados para indicar os identificadores dos *threads* envolvidos.

Os eventos são recebidos pelo monitor e tratados tal como sucede com os eventos produzidos pela própria JVM. Estes eventos permitem assim obter traços de execução com eventos

adicionais, que podem também ser usados para identificar o comportamento de execução relativamente a quaisquer blocos de código específicos ou a passagem por quaisquer outros pontos no código fonte do programa.

A ferramenta de monitorização externa à JVM, que foi também desenvolvida neste trabalho, interactua com esta instrumentação para receber todos os eventos recolhidos. Esta abordagem permite reduzir a complexidade da biblioteca de instrumentação e aliviar a penalização introduzida no desempenho da JVM, ao permitir que parte do processamento seja efectuada em concorrência por um processo externo à JVM. Cada registo passado ao monitor pode ser gravado em ficheiro num formato próprio. Em alternativa, podem estes registos ser formatados como texto e afixados no ecrã durante a execução da própria aplicação.

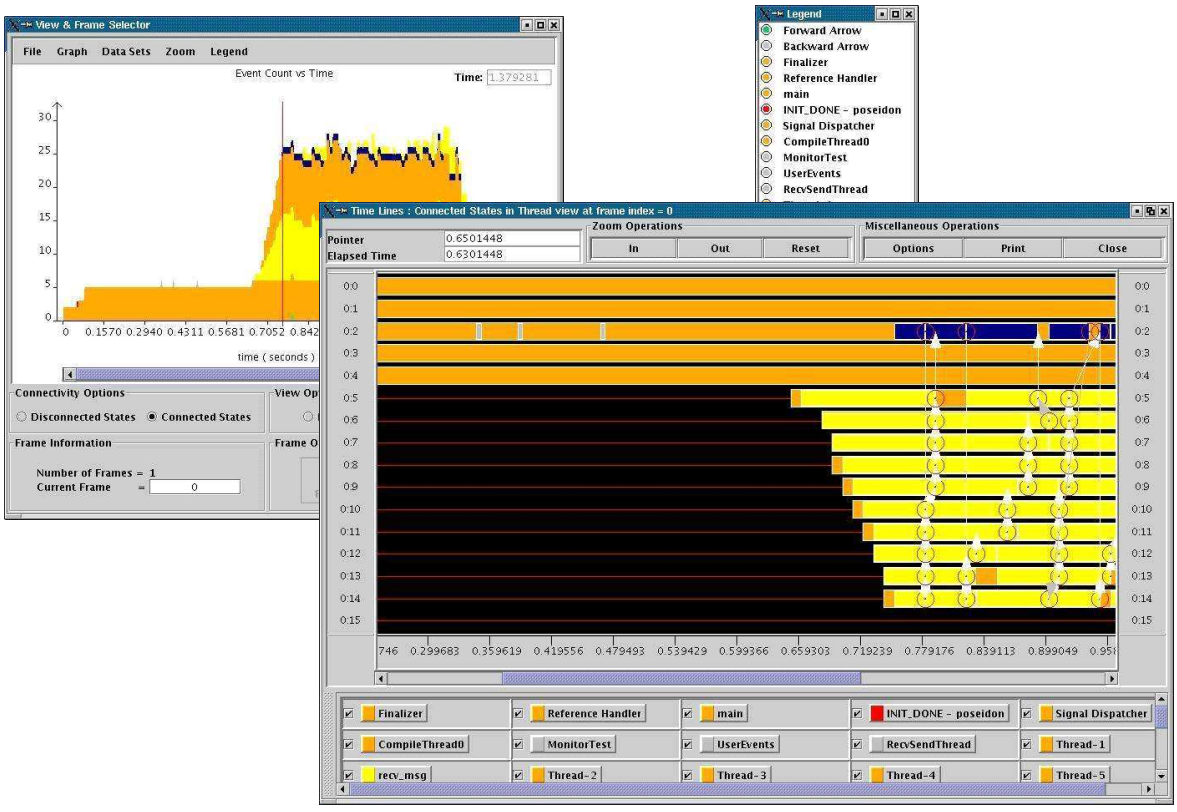

Figura 6.7: Jumpshot mostrando o traço obtido de uma execução Java.

O ficheiro obtido pode ser convertido, *a posteriori*, por uma ferramenta auxiliar, para o formato CLog do MPICH/MPE. Foi, por isso, possível a sua visualização e análise com as ferramentas existentes para esse formato, nomeadamente o Jumpshot (ver fig. 6.7). Nesta conversão, cada *thread* Java é tratado como se fosse um nó de execução na semântica do formato CLog. Os eventos de utilizador introduzidos pelo programador como de interacção

entre dois *threads*, são tratados por este conversor como se fossem trocas de mensagens entre esses nós.

Integração na DAMS

O passo seguinte foi a adaptação da funcionalidade antes apresentada para a monitorização no contexto da DAMS de ambientes distribuídos, permitindo obter o traço de execução de aplicações Java distribuídas. Procurou-se suportar o registo das interacções entre os vários processos, ou seja, entre as várias máquinas JVM. O sistema DAMS desempenha aqui o seu papel de infraestrutura, permitindo ligar a instrumentação de cada JVM a um monitor central responsável pela recolha, e tratamento da informação recolhida (ver figura 6.8). A arquitectura é mais uma vez análoga às anteriores, dando suporte à recolha do traço de execução e permitindo construir um sistema para monitorizar as JVM distribuídas por várias máquinas.

O componente de instrumentação mantém-se inalterado. A recolha dessa informação passa agora a ser efectuada por um monitor semelhante ao anterior, mas adaptado para a sua integração como um sensor na DAMS. Este é agora mais simples, pois não tem de formatar e escrever os registos. É no entanto agora um cliente do serviço de traço local, ao qual se liga de início, e ao qual passa a informação que vai recolhendo. A nova ferramenta de interface com o utilizador, também baseada no código da anterior que formatava e mostrava o traço, recolhe toda a informação que lhe chega. Pode-se continuar a escrever para o ecrã, à medida que a execução decorre, ou gravar em ficheiro, tal como na versão anterior. Caso se pretendam obter os registos por uma ordem total (ou pelo menos, respeitando a causalidade) existe agora a tarefa acrescida de reordenar estes registos.

Por configuração dos serviços de traçado, realizada pelo monitor central a quando do seu arranque, podemos obter duas situações:

1. tal como nos exemplos anteriores, a informação é guardada localmente em cada serviço e, só após a terminação da execução da aplicação, é que essa informação é coligida no monitor central;

2. os serviços de traço não fazem qualquer armazenamento, passando aos seus subscritores cada registo, assim que este lhes chega.

Realizamos assim, sem grandes alterações do sistema de monitorização, a possibilidade

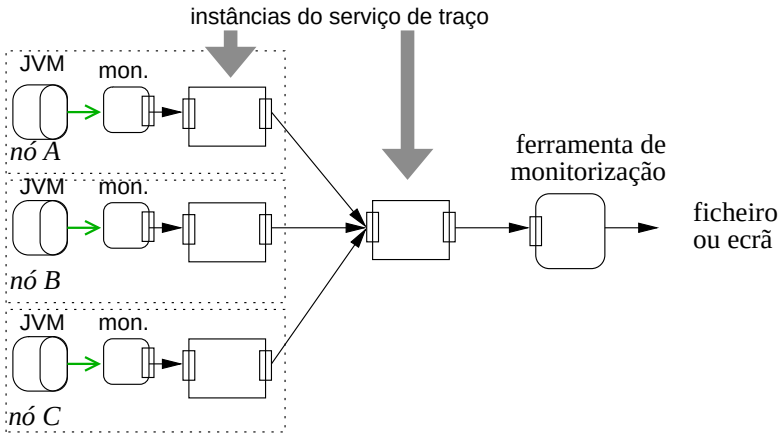

Figura 6.8: Monitorização de várias JVM

de dispor de um sistema para recolha de traços de execução *a posteriori* ou para observar *on-line* a evolução da aplicação.

6.4 Controlo para depuração

Como visto na secção 4.7.2, um serviço para a depuração de processos distribuídos foi desenvolvido no primeiro protótipo da DAMS. O trabalho na área da depuração deste tipo de sistemas foi prosseguido no contexto de outro trabalho de doutoramento [85]). No entanto, um dos primeiros protótipos desenvolvidos de um depurador distribuído foi baseado no primeiro modelo DAMS, tendo essa experimentação contribuído por um lado para validar parcialmente os conceitos do modelo DAMS e, por outro lado, para motivar a evolução do modelo.

O serviço para depuração ou *debugging* aqui apresentado baseia-se na implementação efectuada para a ferramenta PDBG, sobre o primeiro protótipo DAMS. Apresentam-se aqui as operações equivalentes às existentes nesses sistemas e que demonstraram a sua utilidade em várias situações [84, 23, 22, 70]. Note-se que na nova DAMS, existe uma clara separação entre a infraestrutura e os serviços; e que, do ponto de vista da infraestrutura, o serviço central e os vários serviços locais a cada nó não têm suporte diferenciado. Assim, também para as ferramentas, a utilização destas funcionalidades pode ser suportada por um serviço central, ou também utilizando as funcionalidades oferecidas directamente pelos vários serviços locais. O serviço central não necessita de ser o único ponto de acesso, em particular

quando apenas um processo está sob controlo da ferramenta.

A implementação de um sistema equivalente ao desenvolvido no primeiro protótipo divide-se em dois tipos de serviços:

- Um primeiro, o `dbg-local`, que permite controlar e inspeccionar os processos locais a cada nó, por utilização de um depurador sequencial (por exemplo o `gdb`). Este trata-se do apresentado na secção 4.7.2 e desempenha o papel antes desempenhado pelo *driver* em cada nó no primeiro protótipo. Para tal, um processo `gdb` é lançado para cada processo que se pretenda sob o controlo deste serviço.
- Um segundo serviço, central, o `dbg` que, utilizando os anteriores, permite um único ponto de acesso a todos os processos distribuídos no sistema. Este serve também de concentrador de todas as notificações vindas dos vários `dbg-local`, para facilitar a subscrição pelas ferramentas de todos os eventos gerados no sistema. Este serviço desempenha o papel antes suportado pelo SMod do serviço, no primeiro protótipo.

6.4.1 Serviço para depuração

A implementação do serviço de controlo apresentado na secção 4.7.2 e seguindo o protótipo antes desenvolvido, é suportada pela utilização de um programa para depuração (no caso foi usado o `gdb`) como controlador do processo alvo. A ligação entre o serviço e esse programa, é efectuada pela interface por este oferecida, sendo-lhe enviados comandos (texto) e processando as respostas recebidas, do mesmo modo (fig. 6.9). Esta ligação é suportada por uma ligação bidireccional (*input*/*output*) sobre um "pseudo-terminal", que permite a interacção do serviço com os respectivos canais de entrada e saída *standard* do `gdb`. Cada comando suportado pelo serviço é traduzido no respectivo comando do `gdb`.

O sistema de notificações é oferecido sobre o mecanismo de canais de eventos. Cada serviço oferece um novo canal de eventos onde anuncia alterações na execução dos processos: de cada vez que um processo sobre o seu controlo pára (*breakpoint* por exemplo) ou retoma a execução (*continue* por exemplo), um evento descrevendo o acontecimento e o processo envolvido é criado. Uma ferramenta pode estar atenda a estes acontecimentos subscrevendo directamente os canais dos serviços que lhe interessam.

A interface apresentada anteriormente é disponibilizada aos clientes e implementada pelo

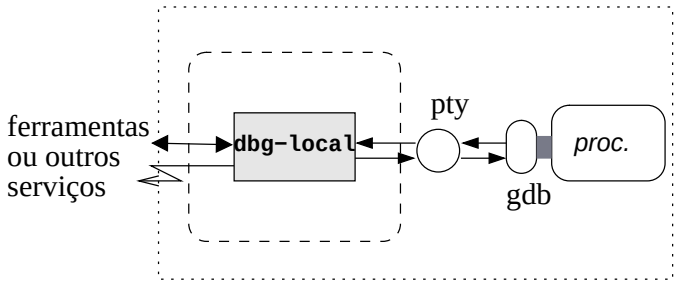

Figura 6.9: Um serviço para depuração usando o `gdb`

respectivo servidor do serviço, da seguinte forma:

`dbgAttach(servID, long pid)`
: permite a ligação do serviço a um processo específico. Corresponde a lançar um processo `gdb` sob o controlo do servidor e efectuar o respectivo comando "attach" ao `pid` indicado;

`dbgDetach(servID, long pid)`
: permite libertar o processo indicado do controlo do serviço. Corresponde a efectuar o "detach" e a terminar o processo do `gdb` respectivo;

`dbgStop(servID, lond pid)`
: neste caso é enviado ao processo indicado, um sinal para parar (SIGSTOP do sistema de operação);

`dbgStep(servID, lond pid)`
: envia ao `gdb` o comando "step";

`dbgContinue(servID, lond pid)`
: envia ao `gdb` o comando "continue";

`dbgSetBreakLine(servID, long pid, char *fname, long line)`
: corresponde a enviar ao `gdb` o comando "break", indicando o ficheiro fonte e linha onde se pretende a paragem;

`dbgDelBreak(servID, long pid, char *fname, long line )`
: corresponde a enviar o comando "clear" ao respectivo `gdb`, para remover o ponto de paragem indicado;

`dbgGetValue(servID, long pid, char *var , char *buf, long bufsz)`
: envia ao respectivo `gdb` o comando "print" onde o argumento é o indicado pela cadeia `var`, sendo a resposta devolvida no vector `buf`;

`dbgSetValue(servID, long pid, char *var, char *buf, long bufsz)`
: corresponde a enviar o comando "set" ao `gdb`, com a variável e respectivo valor dado pelos restantes argumentos.

6.4.2 Serviço central para depuração

O serviço para depuração de aplicações distribuídas, desenvolvido no primeiro protótipo da DAMS, já antes referido, permite a qualquer ferramenta controlar um ou vários processos da aplicação. A arquitectura deste serviço, como é de esperar, reflecte o modelo anterior, apresentado na secção 4.2. Neste, o SMod suporta os diversos comandos disponíveis aos clientes, encaminhando-os para o respectivo *driver* do serviço, com base no identificador do processo alvo. Este mantém informação para cada processo controlado (*attached*), sobre sua localização no sistema, para poder encaminhar cada pedido usando o LM localizado no mesmo nó que o processo alvo, fazendo chegar cada comando ao *driver* devido, como descrito na referida secção. Esta arquitectura é semelhante à de outros sistemas [65, 110], onde também um servidor central coordena representantes remotos em cada máquina, que realmente interactuam com os processos da aplicação.

Dispõe-se assim, de forma transparente e independente das particularidades dos depuradores usados em cada máquina, das funcionalidades para parar, continuar, colocar/remover pontos de paragem, etc. em qualquer/s processo/s da aplicação. Cada *driver* corresponde ao serviço de controlo para depuração apresentado no secção 6.4.1, na qual a descrição anterior se baseou.

As operações então suportadas agora pelo serviço central, são as mesmas do antigo serviço no primeiro protótipo da DAMS, permitindo a compatibilidade funcional nas ferramentas anteriores. Estes comandos correspondem aos do serviço `dbg-local`, antes apresentado, mas onde o identificador do processo alvo referencia, agora, qualquer processo na máquina virtual da DAMS, mais as seguintes operações:

`dbgLocalID dbgAddHost(servID, hostID)`

permite requerer que o respectivo serviço local (`dbg-local`) seja lançado pelo núcleo no nó indicado, passando a ser possível controlar os processos nessa máquina

`dbgLocalID dbgDelHost(servID, hostID)`

liberta o serviço local no nó referido, libertando (*detach*) todos os processos sob o seu controlo.

`dbgProcessNotification(servID, event)`

verifica se existe alguma notificação pendente, recebendo-a no argumento indicado (estas indicam as alterações ao estado dos processos sob depuração).

A utilização do novo modelo da DAMS em substituição do primeiro protótipo, permite mostrar as suas vantagens acrescidas de uma infraestrutura flexível, ao ser capaz de suportar as mesmas funcionalidades do modelo anterior. O serviço `dbg`, a usar pelas ferramentas, serve de intermediário entre as ferramentas e os serviços locais anteriores, permitindo oferecer funcionalidades de âmbito geral ao sistema distribuído, como as existentes no primeiro protótipo da DAMS. Este terá de ser capaz de identificar qualquer processo no sistema alvo e encaminhar os comandos para controlo de cada processo para o serviço local respectivo. Estes identificadores são conseguidos pela junção do identificador local do processo, como é reconhecido pela instância do serviço local no nó respectivo, com o identificador desse serviço local. As listas de processos em cada nó podem ser obtidas por interacção com o respectivo núcleo DAMS.

A figura 6.10 apresenta um esquema da arquitectura do sistema sobre a nova DAMS.

O serviço centralizador cria também o seu próprio canal de eventos que as ferramentas clientes podem subescrever. Este serviço, por seu lado, torna-se subscritor de todos os canais de eventos associados aos vários serviços remotos, tornando-se um concentrador de todas as notificações no sistema. Cada ferramenta pode, subscrevendo apenas o canal de eventos oferecido pelo serviço para depuração central, ser notificada de todas as alterações no estado dos processos sob controlo. É assim possível, neste novo sistema, dispensar a primitiva que antes existia para testar a chegada de notificações (`dbgProcessNotification`) e, em alternativa, usar-se apenas o mecanismo de canais de eventos da nova DAMS.

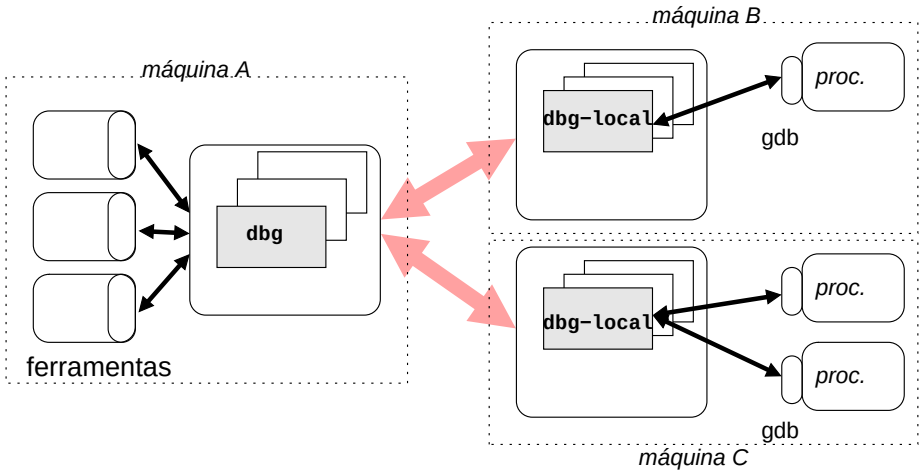

Figura 6.10: Um serviço para depuração global na DAMS

6.5 Controlo dinâmico (*steering*)

No âmbito de um projecto de fim do curso da Licenciatura em Engenharia Informática [124], foi desenvolvida uma aplicação baseada em algoritmos genéticos (AG) [61, 48], com recurso a um sistema já existente, o Sugal [66]. Neste programou-se a optimização de uma solução para um problema de transporte e bombagem de águas, da área da Engenharia do Ambiente, com base num modelo matemático sem solução algorítmica tradicional. Este sistema, foi adaptado para passar a trabalhar em ambiente distribuído segundo um modelo de ilhas comunicantes e passou a incluir um novo componente de *steering* capaz de controlar as várias ilhas. Este componente permite também ao utilizador monitorizar alguns indicadores que representam a evolução do algoritmo e ajustar alguns parâmetros de configuração do algoritmo, numa tentativa de o fazer convergir mais rápidamente para uma solução satisfatória, para cada problema em causa.

Cada ilha é um processo autónomo que implementa o AG. Este possui um conjunto de variáveis que estão declaradas como parâmetros do AG, e outro que representa a qualidade da melhor solução obtida até à geração corrente.

O componente de *steering* é um processo único e centralizador que, por interacção directa com o utilizador, comunica com todos os processos do sistema e assim pode observar e controlar todos eles.

Nesta arquitectura, um componente satélite a cada processo de AG, permite tornar in-

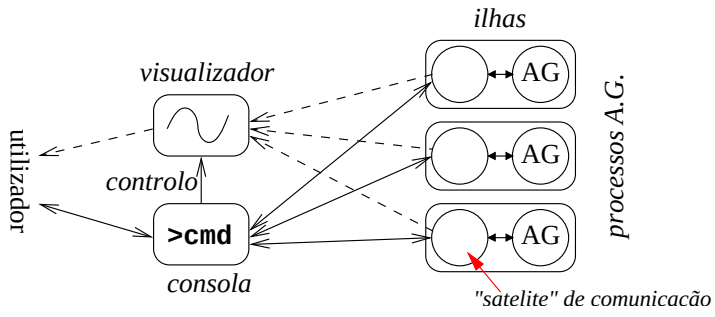

Figura 6.11: Arquitectura do sistema de *steering* dedicado

dependentes a comunicação e o ciclo de cálculo, tornando-os assíncronos. Este satélite é responsável por receber todos os comandos e pedidos de migrações, enquanto o processo de cálculo das novas gerações decorre e, de forma inversa, pelo envio para os destinatários das gerações que migram ou as respostas aos comandos.

Para a implementação deste sistema foi necessário adaptar o código original do Sugal nos seguintes aspectos (ver fig. 6.11):

- o componente que implementa o algoritmo genético foi tornado autónomo, podendo executar como uma ilha (onde o algoritmo evolui autonomamente) e interagindo com o resto do sistema por um processo satélite, responsável por todas as comunicações;
- as variáveis que descrevem a evolução do algoritmo e os parâmetros que o controlam foram tornados remotamente acessíveis via esses satélites;
- o sistema gráfico de visualização foi separado e tornou-se possível, sob o controlo do componente de *steering*, ligá-lo dinâmicamente a qualquer ilha.

O resultado final foi um sistema que permite ao utilizador experiente observar e controlar as várias ilhas, incluindo “guiá-las” tentando que a convergência para a solução seja mais rápida, o que se torna bastante benéfico nas resoluções de problemas complexos que exijam bastante tempo de execução e que dependem criticamente de determinados parâmetros.

Cada ilha coordena-se com o respectivo satélite após gerar cada nova geração. Nesse instante são tratados os comandos externos vindos da consola de controlo ou das outras ilhas, permitindo as seguintes operações:

- parar ou continuar a execução do algoritmo;

- consultar e alterar os parâmetros relevantes;
- consultar os valores indicadores da qualidade da solução;
- aceitar novos elementos para a população do AG (migrantes);
- passar a interagir com o visualizador (fig. 6.12).

O ciclo principal de cada ilha corresponde, de uma forma simples, ao seguinte pseudo-código:

```
while( running )
{
   setParameters()
   runOneGeneration()
   getParameters()
   if ( connectedToVisualizer ) sendParameters()
   if ( commandPending ) processComand()
   if ( migrationTime ) migrate()
   if ( reachedFitValue ) waitComands()
}
```

Esta arquitectura permitiu, por um lado, obter uma solução que paraleliza a geração de novas soluções (as gerações), utilizando vários processos em paralelo. Permite também que, após um determinado número de iterações, existam "migrações" entre as ilhas, retomando-se de seguida o algoritmo habitual em cada uma. É assim possível o cruzamento entre as soluções que estão sendo obtidas em cada ilha, provavelmente segundo parâmetros diferentes.

Esta solução necessitou do desenvolvimento de uma infraestrutura para observação e controlo das ilhas e de um protocolo específico entre o componente de *steering* e os restantes processos. Esta foi desenhada para uma plataforma de troca de mensagens. Neste trabalho foram desenvolvidas duas versões, uma usando o PVM como suporte à comunicação e a outra usando o MPI.

Posteriormente foi desenvolvida uma nova versão deste sistema de *steering* mas utilizando a DAMS como infraestrutura. Tal foi conseguido por adaptação da versão PVM do sistema de AG e pela concepção e implementação no primeiro protótipo da DAMS, de um novo serviço responsável pela observação e controlo dos AG.

Tirando partido da infraestrutura oferecida no primeiro protótipo DAMS sobre o PVM, o componente de *steering* da aplicação foi reimplementado integrando-se como um serviço

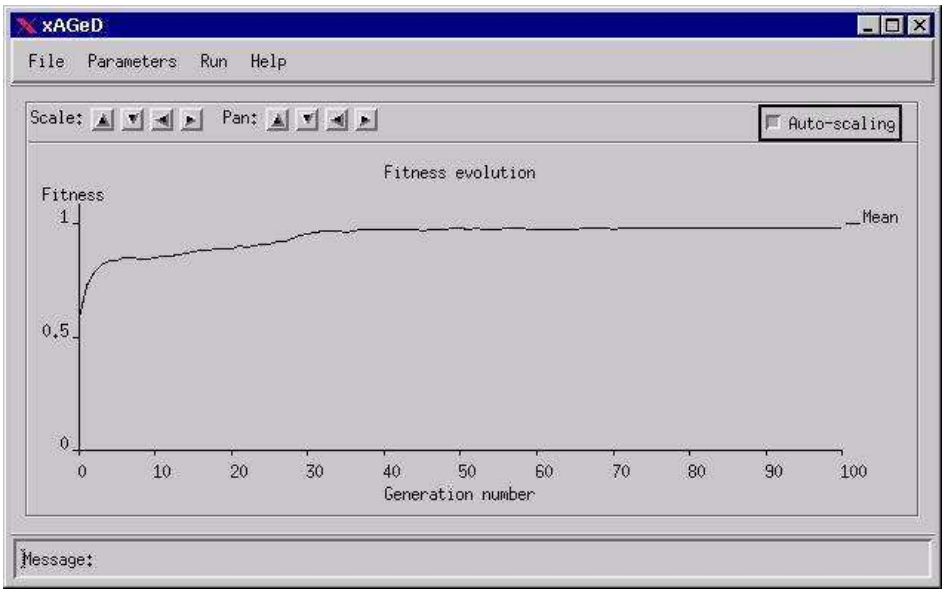

Figura 6.12: Aspecto do visualizador de *steering* ligado a uma ilha.

DAMS. O objectivo foi conseguir o mesmo tipo de consola de *steering*, que acede de forma transparente a cada uma das ilhas para consulta dos parâmetros exportados e permitindo a alteração de cada um destes. Os seguintes aspectos foram tidos em conta:

- disponibilizar um serviço com uma interface genérica para o controlo e observação das ilhas que implementam o AG;
- manter as funcionalidades na consola de interface com o utilizador e tornar estas independentes do sistema de AG (tornar adaptável a outros AG);
- procurar manter inalterado o código correspondente à implementação dos AG nas ilhas e respectivas migrações entre ilhas sobre o PVM;
- o sistema DAMS e seus serviços não podem interferir com o normal funcionamento do AG.

Na solução utilizando a DAMS, tirou-se partido da interface disponibilizada pelas ilhas na implementação de um módulo (`dga-driver`), que é o *driver* do serviço DGA, acessível pelo *Service Module* no SM. Este desempenha agora o papel que antes era da responsabilidade do "satélite". Por seu lado, a ferramenta desenvolvida, ou seja, a consola com o utilizador, utiliza assim a DAMS como suporte à distribuição da aplicação, controlando o

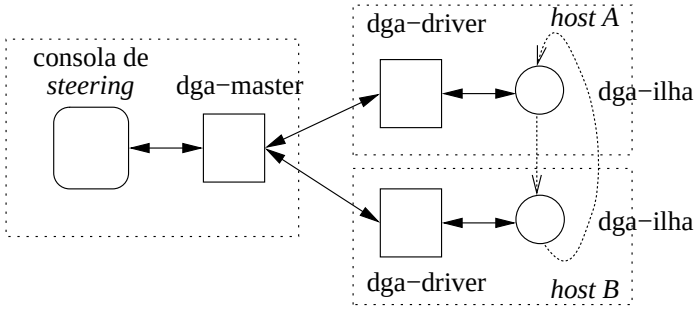

Figura 6.13: *Steering* de AG utilizando o primeiro modelo da DAMS

lançamento das ilhas, configuração, observação e avaliação das soluções. As funcionalidades implementadas pelo serviço `dga` são utilizadas na implementação de um conjunto de comandos acessíveis ao utilizador, idênticos aos antes disponíveis. Estas são operações suportadas pelo serviço DGA na DAMS:

dgaRun — é responsável por lançar uma ilha sob o controlo de um `dga-driver` (esta relação é de uma ilha para um *driver* neste modelo da DAMS). Este lançamento é suportado pela gestão de recursos do PVM (`pvm_spawn`), mas desencadeado pelo respectivo *driver*.

dgaStop, dgaContinue — enviam os comandos, respectivamente, para parar ou retomar a execução do AG.

dgaGet, dgaSet — permitem consultar ou alterar, respectivamente, os parâmetros e variáveis disponibilizados pela ilha.

dgaList — lista os identificadores das ilhas. Estes são na realidade os *taskid* dos respectivos satélites com os quais os *drivers* interactuam.

O serviço em si contempla que os comandos sejam efectuados sobre qualquer ilha sob o seu controlo ou, por indicação de um identificador especial (`all`), sobre todas as ilhas, distribuindo o mesmo comando sobre todos os *drivers* em execução, para que estes os apliquem às respectivas ilhas.

No caso particular desta solução, algum cuidado foi colocado na utilização da plataforma PVM, pelo problema já referido na secção 5.2.2. As ilhas organizam-se em anel para a transferência de migrantes entre si. Foi necessária uma pequena alteração no código da

ilha para que estas funcionassem com base num grupo PVM a quando dessa organização, para que não considerassem as *tasks* do sistema DAMS e da ferramenta de controlo, como participantes nesse anel (estas não são ilhas!).

O uso da DAMS facilitou assim, como pretendido, o desenvolvimento deste tipo de sistemas e ferramentas. Existem no entanto algumas desvantagens na arquitectura do primeiro protótipo da DAMS usado no sistema para a distribuição da aplicação, que são ultrapassadas com o novo modelo da DAMS:

- a ligação directa entre a ferramenta e a aplicação utilizando a mesma plataforma que suporta a aplicação, tornava difícil aplicar a ferramenta a uma situação idêntica mas onde a plataforma seja outra;

- apesar da monitorização efectuada, não era fácil dar-lhe novas utilizações, como sejam, obter o historial dos parâmetros observados durante a execução, usar esta informação noutra ferramenta (p.e. um agente que "guie" o AG no lugar do utilizador), etc.;

- o sistema original não pode ser usado para suportar outras observações ou formas de controlo de que outras ferramentas necessitem;

- o sistema dificilmente era adaptável a outros ambientes/aplicações de algoritmos genéticos.

Na nova DAMS, o modelo permite que, o que antes era um *driver* seja agora também um serviço da DAMS. É por isso possível que a ferramenta contacte estes serviços directamente quando necessário. O serviço DGA pode manter na sua interface todas as suas funcionalidades, obtendo-se assim um sistema, nesse aspecto equivalente ao anterior. No entanto agora não teria sido necessário alterar a ilha para considerar um grupo PVM. Tirando partido do serviço de traçado, é possível implementar um sistema de recolha e visualização *on-line* (e também para tratamento *post-mortem*), da evolução dos AG, sem necessidade de a ferramenta testar periodicamente cada ilha, através da consola de comandos interactiva.

Nesta nova configuração pode-se utilizar uma arquitectura semelhante às antes vistas para o MPI e Java. Cada ilha a cada volta do ciclo do AG (ou a cada N iterações) produz um registo, que é introduzido no respectivo serviço de traçado, descrevendo o estado da computação (fig. 6.14). Esta informação pode ir de imediato para uma ferramenta que visualiza

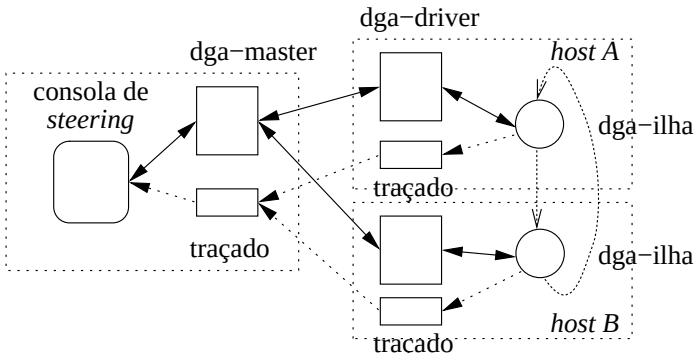

Figura 6.14: *Steering* de AG utilizando o novo modelo da DAMS

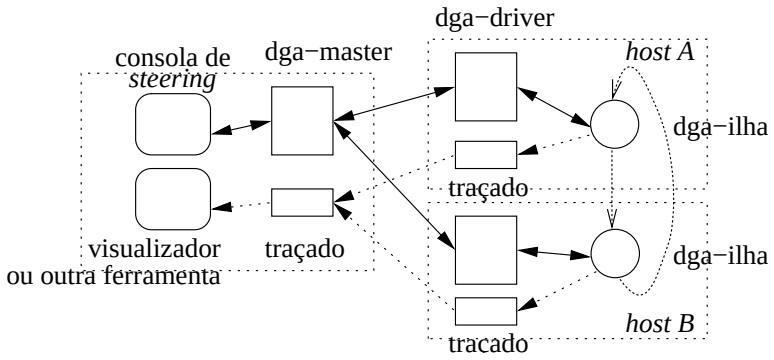

Figura 6.15: *Steering* de AG utilizando a DAMS (com visualizador autónomo)

a evolução dos AG, ou pode ser guardada para posterior processamento, tal como nos casos anteriores de PVM-Prolog, MPI e Java, quer para observação *on-line* ou para obter um registo da evolução do sistema após a sua terminação (fig. 6.15).

Outra abordagem para outra solução, poderia passar por uma arquitectura que aproveitasse os serviços já existentes. Este caso consiste numa situação onde se juntam os dois aspectos, o de observação e o de controlo. Tal pode ser conseguido com utilização das funcionalidades já oferecidas pelo serviço de depuração, podendo ainda recorrer ao serviço de traçado para a recolha dos valores observados ao longo do tempo (traço da evolução), em cada ilha. Esta abordagem teria a vantagem de não necessitar de reimplementar a infraestrutura de comunicação, e de não necessitar de alterações ao código do programa que implementa a ilha. Para tal usar-se-iam as operações para consulta e alteração do valor das variáveis pretendidas, no caso, as que são os parâmetros do AG. O controlo da execução para consulta e alteração desses valores, seria feito à custa da introdução de pontos de paragem,

de forma semelhante à que se usa para a depuração de programas. Estas novas funcionalidades podem ser implementadas como um novo serviço. Seria este novo serviço que, usando o de depuração, interagia com os elementos da aplicação (as ilhas), podendo implementar as operações específicas semelhantes à interface que antes era oferecida pelo *driver* do `dga` ou pelo satélite dos primeiros protótipos.

6.6 Conclusão

Em comparação com outras abordagens, em particular OMIS [88] e FIRST [111], ressaltam, como principais diferenças:

OMIS a abordagem seguida foi a de propor uma interface normalizada, baseada em operações genéricas simples, que permitam suportar as mais variadas ferramentas. No modelo DAMS, definiu-se uma arquitectura abstracta flexível, podendo esta suportar serviços, incluindo um que permita implementar a interface da norma OMIS;

FIRST baseia-se no sistema CORBA como infraestrutura que lhe garante a flexibilidade. Fica no entanto totalmente dependente desse sistema. A interacção com os processos alvo, é exclusivamente baseada no sistema DynInst [13]. Na abordagem DAMS, isolam-se as definições dos serviços em relação à plataforma de base, permitindo implementações da DAMS sobre outros ambientes; é da responsabilidade da realização de cada serviço, a decisão de como interactuar com o alvo da monitorização, com recurso aos mais adequados mecanismos de instrumentação.

O modelo *DAMS*, cuja génese data de 1998, serviu de base aos desenvolvimentos experimentais, além dos aqui reportados, efectuados no contexto de diversos projectos de investigação [23, 22, 28, 116, 112, 108, 33, 39], que contribuíram para validar o conceito de uma arquitectura de monitorização orientada para os serviços. Entretanto, desenvolvimentos paralelos na área dos sistemas distribuídos, vieram a conduzir a diversas propostas de modelos de arquitecturas orientadas para serviços que caracterizam a maioria dos sistemas actuais.

A experimentação efectuada em torno do modelo DAMS comprovou plenamente os objectivos iniciais deste trabalho. A flexibilidade desta abordagem foi reconhecida pela comunidade internacional [123], numa comparação sistemática deste modelo com o modelo

OMIS e a proposta FIRST, bem como em apresentações em conferências [29, 24, 30] e revistas [41].

Capítulo 7

Conclusões

Nesta dissertação apresentou-se uma proposta de um modelo de arquitectura para suportar um conjunto variado de funcionalidades de observação e controlo de aplicações paralelas e distribuídas. Os objectivos prioritários prendem-se com as fases de desenvolvimento para a detecção de erros e avaliação de desempenho, assim como, durante o funcionamento da aplicação, a observação da sua evolução e possível controlo dinâmico do seu comportamento.

7.1 Avaliação

A necessidade deste tipo de infraestrutura foi comprovada por várias experiências em vários ambientes de utilização. O modelo proposto apresenta as seguintes vantagens sobre os tradicionais sistemas, específicos a cada plataforma ou ferramenta:

- esta infraestrutura apresenta vários níveis que permitem vários graus de separação entre as funcionalidades oferecidas e a plataforma que as suporta;
- o suporte à modularidade, através da organização em serviços, permite que a interface oferecida às ferramentas possa abstrair dos detalhes da sua implementação e das dependências relativamente à plataforma base;
- a extensibilidade, conseguida de forma incremental sobre os serviços já existentes, permite explorar novas funcionalidades tirando partido das já existentes;
- o serviço de traço procura servir de suporte a vários tipos de monitorização, permi-

tindo assim enriquecer a infraestrutura para facilitar a rápida implementação de novas formas de observação e controlo;

- torna mais viável a experimentação da monitorização de novas plataformas de suporte às aplicações, sistemas ou modelos de programação;
- torna mais viável a reutilização de ferramentas existentes nessas novas aplicações ou a experimentação de novas ferramentas sobre a infraestrutura de monitorização existente;
- oferece uma plataforma partilhada pelas várias ferramentas, evitando a repetição de funcionalidades e consequente sobre-utilização de recursos, e a disparidade entre os estados observados por cada ferramenta;
- esta plataforma pode servir ao suporte de mecanismos para cooperação entre as ferramentas, quer pela partilha de estado (da aplicação ou da infraestrutura de monitorização e das próprias ferramentas), quer pela possibilidade de implementar mecanismos para sincronização e/ou interacção entre as ferramentas.

7.2 Direcções de trabalho futuro

Consideram-se as seguintes direcções de trabalho futuro:

- aplicação da infraestrutura desenvolvida para suportar as actividades de observação e controlo, para suportar a avaliação do desempenho e o controlo dinâmico em sistemas de computação paralela e distribuída;
- nos vários cenários de aplicação, avaliar a eficiência e sobrecarga computacional dos protótipos;
- em particular, está em curso a aplicação da DAMS para a monitorização de um sistema de simulação paralela no domínio da Engenharia do Ambiente, baseado no modelo do comportamento de fluxos de poluentes marítimos;
- estudo da evolução do modelo DAMS para suportar a monitorização no contexto de ambientes integrados para a resolução de problemas (*Problem Solving Environments*), em plataformas distribuídas baseadas em *Grid Computing*, em particular explorando

os requisitos de observação e controlo de sistemas dinâmicos, isto é, apresentando grandes variações no comportamento das aplicações e dos recursos computacionais que as suportam;

- integração da arquitectura da DAMS em plataformas de suporte a *Grid Computing*;
- aplicação do modelo da DAMS para o desenvolvimento de sistemas de monitorização integrados, combinando as abordagens de observação e análise *post mortem* e monitorização *online*, para optimização do desempenho de aplicações distribuídas dinâmicas.

Bibliografia

[1] M. Abrams. *An Example of Deriving Performance Properties from a Visual Representation of Program Execution*. *IEEE Transactions on Parallel and Distributed Systems*, 8(6), Jun. 97.

[2] *APART – Automatic Performance Analysis: Resources and Tools, Esprit Working Group*. `http://www.fz-juelich.de/apart-1/`.

[3] Y. Arrouye. *Environnements de Visualisation pour l'Évaluation des Performances des Systèmes Paralléles: étude, conception et réalisation*. Tese de Doutoramento, Institut National Polytechnique de Grenoble, Nov. 95.

[4] R. A. Aydt. *SDDF: The Pablo Self-Describing Data Format*. Relatório técnico, Department of Computer Science, University of Illinois, Abr. 1994.

[5] O. Babaoglu, et al. *Paralex: An Environment for Parallel Programming in Distributed Systems*. In *Proc. 1992 Int'l Conf. Supercomputing*, pág. 178–187, N.Y., July 92. ACM Press.

[6] O. Babaoglu, K. Marzullo. *Consistent Global States of Distributed System: Fundamental Concepts and Mechanisms*. In S. Mullender, editor, *Distributed Systems*. Addisson-Wesley, 1993.

[7] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, K. Moore. *HeNCE: a heterogeneous network computing environment*. *Scientific Programming*, 3(1):49–60, Spring 1994.

[8] A. Beguelin, G. A. Geist, J. Dongarra, K. Moore, P. Newton, R. Manchek, V. Sunderam. *HeNCE: A Users' Guide Version 2.0*, Nov. 18 1997. http://www.netlib.org/hence/HeNCE-2.0-doc.ps.gz.

[9] A. L. Beguelin. *Xab: A Tool for Monitoring PVM Programs*. In *Proceedings Workshop on Heterogeneous Processing*, pág. 92–97. IEEE Computer Society Press, Abr. 93.

[10] A. L. Beguelin, J. Dongarra, A. Geist, V. Sunderam. *Visualization and Debugging in a Heterogeneous Environment*. *IEEE Computer*, pág. 88–95, Jun. 1993.

[11] T. Bemmerl, P. Braun. *Visualization of Message Passing Parallel Programs with TOPSYS Parallel Programming Environment*. *Journal of Parallel and Distributed Computing*, (18):118–128, 1993.

[12] S. Browne, J. Dongarra, N. Garner, G. Ho, P. Mucci. *A Portable Programming Interface for Performance Evaluation on Modern Processors*. *The International Journal of High Performance Computing Applications*, pág. 189–204, Fall 2000.

[13] B. Buck, J. K. Hollingsworth. *An API for Runtime Code Patching*. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[14] R. Butler, E. Lusk. *User's Guide to the p4 Programming System*. Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue Argonne, IL 60439-4801, anl-92/17 edition, October 1992.

[15] W. Cai, W. J. Milne, S. J. Turner. *Graphical Views of the Behavior of Parallel Programs*. *Journal of Parallel and Distributed Computing*, 18(2):223–230, 1993.

[16] T. L. Casavant. *Tools and Methods for Visualization of Parallel Systems and Computations*. *Journal of Parallel and Distributed Computing*, 18(2):103–104, 1993. Guest editor's introduction.

[17] A. Chan, W. Gropp, E. Lusk. *User's Guide for MPE extensions for MPI programs*. Relatório técnico, Argonne National Laboratory, 1998.

[18] C. M. Chase, A. L. Cheung, A. P. Reeves, M. R. Smith. *Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures*. *Journal of Parallel and Distributed Computing*, 16(2):79–91, 1992.

[19] A. P. Cláudio, J. D. Cunha, M. B. Carmo. *MPVisualizer: A General Tool to Debug Message Passing Parallel Applications.* In *HPCN Europe*, pág. 1199–1202. Springer-Verlag, 1999.

[20] C. Clémençon, J. Fritcher, M. J. Meehan, R. Rühl. *An Implementation of Race Detection and Deterministic Replay with MPI*. Relatório técnico CSCS TR-95-01, Swiss Scientific Computing Center (CSC–ETH), January 1995.

[21] C. Clémençon, J. Fritcher, R. Rühl. *Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool*. Relatório técnico CSCS TR-94-09, Swiss Scientific Computing Center (CSC–ETH), November 1994.

[22] Copernicus Programme. *High Performance Computing Tools for Industry (HPCTI)*, final report,university of westminster edition, 1996.

[23] Copernicus Programme. *Software Engineering for Parallel Processing (SEPP)*, final report,university of westminster edition, 1997.

[24] J. C. Cunha, V. Duarte. *Monitoring PVM Programs Using the DAMS Approach*. In *5th Euro PVM/MPI*, vol. 1497 de *LNCS*, pág. 273–280. Springer-Verlag, 1998.

[25] J. C. Cunha, V. Duarte, J. Lourenço, T. Antão. *Monitoring and Debugging Support*. HPCTI Project 3rd Progress Report, Copernicus Programme, University of Westminster, 1996.

[26] J. C. Cunha, V. Duarte, J. Lourenço, P. Medeiros. *A Software Architecture for the Integration of Monitoring, Debugging, and Profiling Tools for Parallel Program Development*. Relatório técnico, Departamento de Informática, FCT-Universidade Nova de Lisboa, Nov. 1997.

[27] J. C. Cunha, J. Lourenço, T. Antão. *An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger*. *Journal of Systems Architecture*, 45(11):897–907, 1999. Elsevier Science Press.

[28] J. C. Cunha, J. Lourenço, V. Duarte. *Tool Integration Issues for Parallel and Distributed Debugging*. In *3rd SEIHPC Copernicus Workshop, Madrid*. University of Westminster, London, Jan. 1998.

[29] J. C. Cunha, J. Lourenço, J. Vieira, B. Moscão, D. Pereira. *A Framework to Support Parallel and Distributed Debugging*. In *Proc. of the International Conference on High-Performance Computing and Networking (HPCN'98)*, vol. 1401 de *LNCS, Springer-Verlag*, pág. 708–717, Amsterdam, 1998.

[30] J. C. Cunha, P. Medeiros, V. Duarte, J. Lourenço, M. C. Gomes. *An Experience in Building a Parallel and Distributed Problem–Solving Environment*. In *PDPTA'99 – International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. IV, pág. 1804–1809. CSREA Press, Jul. 1999.

[31] J. C. Cunha, P. Medeiros, J. Lourenço, V. Duarte, J. Vieira, B. Moscão, D. Pereira, R. Vaz. *The DOTPAR Project: Towards a Framework Supporting Domain Oriented Tools for Parallel and Distributed Processing*. In *International Conference on High Performance Computing and Networking (HPCN'98)*, vol. 1401 de *LNCS*, pág. 952–954. Springer, Abr. 1998.

[32] J. C. Cunha, R. F. P. Marques. *Distributed Algorithm Development with PVM-Prolog*. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, London*, 1997.

[33] J. C. Cunha, P. Kacsuk, S. Winter, editores. *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments*, vol. 5 de *Series on Advances in Computation: Theory and Practice*. Nova Science Publishers Inc., 2000.

[34] J. C. Cunha, J. Lourenço, V. Duarte. *Using DDBG to Support Testing and High-level Debugging Interfaces*. *Computers and Artificial Intelligence*, 17(5):429–439, 1998.

[35] J. C. Cunha, J. Lourenço, V. Duarte. *The DDBG Distributed Debugger*. In *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments*, Capítulo 13. Nova Science Publishers Inc., 2000.

[36] J. C. Cunha, J. Lourenço, V. Duarte. *Debugging of Parallel and Distributed Programs*. In *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments*, Capítulo 5. Nova Science Publishers Inc., 2000.

[37] F. Darema. *Parallel Applications Performance Methodology*. In M. Simmons, R. Koskela, I. Bucher, editores, *Instrumentation for Future Parallel Computing Systems*, Capítulo 3, pág. 49–57. ACM Press/Addison-Wesley PC, New York, 1989.

[38] T. Delaitre, G. Justo, F. Spies, S. Winter. *EDPEPPS : An Environment for the Design and Performance Evaluation of Portable Parallel Software*. UKPAR Conference, 1996., 1996.

[39] *Depurador Visual de Programas baseados em Threads*, 1998–2001. coop. bilateral com LMC-IMAG/INRIA, Grenoble.

[40] V. Duarte, J. C. Cunha. *Using Tape/PVM for Monitoring GRAPNEL Programs*. In *HPCTI Project, COPERNICUS Programme, Final Report*. University of Westminster, London, Out. 1996.

[41] V. Duarte, J. Lourenço, J. C. Cunha. *Supporting On-Line Distributed Monitoring and Debugging*. *Jornal of Parallel and Distributed Computing Practices, Special Issue on Monitoring Systems and Tool Interoperability*, 4(4):261–274, 2001.

[42] M. Ducassé. *OPIUM: An Extendable Trace Analyser for PROLOG*. Relatório técnico 1127, IRISA, Rennes-France, Sep. 97. Projet LANDE.

[43] H. Dörr. *STED - A Small General-Purpose Tool for Software Monitoring*. In R. Grebe, et al., editores, *Transputer Applications and Systems '93*, pág. 410–420. Institut für Informatik, Freie Universität Berlin, IOS Press, 1993.

[44] *EuroTools – Working Group for European HPCN Tools Promotion*. `http://www.irisa.fr/cgi-bin/cgiwrap/pazat/pazat_dir/swish/tools/ /Tools.pl`.

[45] T. Fahringer. *Estimating and Optimizing Performance for Parallel Programs*. *IEEE Computer*, 28(11):47–56, Nov. 95.

[46] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, 1994.

[47] A. P. Goldberg, A. Gopal, A. Lowry, R. Strom. *Restoring Consistent Global States of Distributed Computations*. *ACM SIGPLAN Notices*, pág. 144–154, Dez. 1991.

[48] D. E. Goldberg. *Genetic Algorithms: in Search Optimization & Machine Learning*. Addison Wesley, Reading, MA, 1989.

[49] H. A. Goosen, P. Hinz, D. W. Polzin. *Experience Using the Chiron Parallel Program Performance Visualization System*. Relatório técnico, CSD - University of Cape Town, 1995.

[50] S. Graham, P. Kessler, M. McKusick. *An Execution Profiler for Modular Programs*. *Software - Practice and Experience*, (13):671–685, 1983.

[51] W. Gropp, E. Lusk, N. Doss, A. Skjellum. *A high-performance, portable implementation of the MPI message passing interface standard*. *Parallel Computing*, 22(6):789–828, Set. 1996.

[52] W. D. Gropp, E. Lusk. *User's Guide for* `MPICH`, *a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[53] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, N. Mallavarupu. *Falcon: on-line monitoring and steering of large-scale parallel programs*. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*, pág. 422–429. McClean, 1995.

[54] G. Haring, C. Lindemann, M. Reiser, editores. *Performance Evaluation: Origins and Directions*. Springer-Verlag, 2000.

[55] M. Heath, J. Finger. *ParaGraph: A Performance Visualization Tool for MPI*. Univ. of Illinois and Univ. of Tennesse, 1999.

[56] M. T. Heath, J. A. Etheridge. *Visualizing the Performance of Parallel Programs*. *IEEE Software*, pág. 29–39, Set. 1991.

[57] M. T. Heath, J. A. Etheridge. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*. Relatório técnico, U. Illinois, Oak Ridge Nac. Lab., Jun. 1992.

[58] M. Heath, A. Malony, D. Rover. *The Visual Display of Parallel Performance Data*. *IEEE Computer*, 28(11):21–28, Nov. 95.

[59] J. Henriques. *Monitorização de aplicações Java*. Relatório técnico, Dep. de Informática, FCT/UNL, 2002. Projecto de fim de curso.

[60] H. Hersey, S. Hackstadt, L. Hansen, A. Malony. *Viz: A Visualization Programming System*. Relatório técnico CIS-TR-96-05, DCIS-University of Oregon, Abr. 96.

[61] J. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[62] J. K. Hollingsworth, B. Buck. *DyninstAPI Programmer's Guige*. CSD/University of Maryland, release 1.2 edition, Set. 1998.

[63] J. K. Hollingsworth, B. P. Miller, J. Cargille. *Dynamic Program Instrumentation for Scalable Performance Tools*. In *Proc. of Scalable High-Performance Computing Conf.*, 1994.

[64] J. K. Hollingsworth, J. James E. Lumpp, B. P. Miller. *Performance Measurement of Parallel Programs*. In *ISIPCALA '93*, pág. 239–254, 1993.

[65] R. Hood. *The p2d2 Project: Building a Portable Distributed Debugger*. In *Proceedings of the* 2^{nd} *Symposium on Parallel and Distributed Tools (SPDT'96)*, Philadelphia PA, USA, 1996. ACM.

[66] A. Hunter. *SUGAL User Manual V2.1*. University of Sunderland, 1995.

[67] Institut für Informatik der TU München. *TOPSYS User's Overview - Ver. 1.0*, Dec 1990.

[68] C. Jeffery, W. Zhou, K. Templer, M. Brazell. *A Lightweight Architecture for Program Execution Monitoring*. *ACM SIGPLAN Notices*, pág. 67–74, Jul. 98.

[69] P. Kacsuk, J. C. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, T. Antão. *A Graphical Development and Debugging Environment for Parallel Programs*. *Parallel Computing*, 22(1997):1747–1770, 1997.

[70] P. Kacsuk, G. Dózsa, T. Fadgyas. *Designing Parallel Programs by the Graphical Language GRAPNEL*. *Microprocessing and Microprogramming*, (41):625–643, 1996.

[71] J. C. Kergommeaux, B. O. Stein. *Pajé: An Extensible Environment for Visualizing Multi-threaded Programs Executions*. In *Proc. Euro-Par 2000*, vol. 1900 de *LNCS*, pág. 133–140. Springer, 2000.

[72] J. M. Kewley, R. Prodan. *A Distributed Object-Oriented Framework for Tool Development*. In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*, pág. 353–362. IEEE Computer Society Press, Jul. 2000.

[73] M. Khouzam, T. Kunz. *Single Stepping in Event-Visualization Tools*. Relatório técnico, DCS-University of Waterloo, 96.

[74] D. Kimelman, P. Mittal, E. Schonberg, P. F. Sweeney, K.-Y. Wang, D. Zernik. *Visualizing the Execution of High Performance Fortran (HPF) Programs*. Relatório técnico, Thomas J Watson RC/IBM, 94.

[75] D. Kimelman, B. Rosenburg, T. Roth. *Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior*. Relatório técnico, RC-IBM, Jul 94.

[76] J. A. Kohl, G. A. Geist. *The PVM 3.4 Tracing Facility and XPVM 1.1*. In H. El-Rewini, B. D. Shriver, editores, *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences (HICSS-29)*, pág. 290–299. IEEE Computer Society Press, 1996.

[77] H. Krawczyk, B. Wiszniewski. *Interactive Testing Tool for Parallel Programs*. In I. Jelly, I. Gorton, P. Crolll, editores, *Software Engineering for Parallel and Distributed Systems*, pág. 98–109, London, UK, 1996. Chapman & Hal.

[78] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. *Communications of the ACM*, 21(7):558–565, Jul. 1978.

[79] D. Lange, Y. Nakamura. *Object-Oriented Program Tracing and Visualization*. *IEEE Computer*, 30(5):63–70, May 97.

[80] T. J. LeBlanc, J. M. Mellor-Crummey. *Debugging Parallel Programs with Instant Replay*. *IEEE Transactions on Cumputers*, C–36(4):471–482, Abr. 1978.

[81] J. Lourenço, J. C. Cunha. *The PDBG Process-level Debugger for Parallel and Distributed Programs*. Relatório técnico, Departamento de Informática, FCT-Universidade Nova de Lisboa, 1998.

[82] J. Lourenço, J. C. Cunha. *Replaying Distributed Applications with RPVM*. In *Proceedings of the* 2^{nd} *Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'98)*, Budapest, Hungary, Out. 1998. Report Series of the Institute of Applied Computer Science and Information Systems, University of Vienna.

[83] J. Lourenço, J. C. Cunha. *Flexible Interface for Distributed Debugging (Library and Engine): Reference Manual (V 0.3.1)*. Departamento de Informática da Universidade Nova de Lisboa, Portugal, Dez. 2000.

[84] J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, B. Wiszniewsk. *An Integrated Testing and Debugging Environment for Parallel and Distributed Programs*. In *Proceedings of the 23rd EUROMICRO Conference (EUROMICRO'97)*, pág. 291–298, Budapeste, Hungary, 1997. IEEE Computer Society Press.

[85] J. Lourenço. *A Debugging Engine for Parallel and Distributed Programs*. Tese de Doutoramento, Departamento de Informática, FCT-Universidade Nova de Lisboa, 2003. A publicar.

[86] T. Ludwig, R. Wismueller, M. Oberhuber. *OCM — An OMIS Compliant Monitoring System*. *EuroPVM/MPI, Lecture Notes in Computer Science, Springer-Verlag*, 1156, 1996.

[87] T. Ludwig, R. Wismüller, A. Bode. *Interoperable Tools Based on OMIS*. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pág. 155–155. ACM Press, 1998.

[88] T. Ludwing, R. Wismüller, V. Sunderam, A. Bode. *OMIS – On-line Monitoring Interface Specification*. Relatório técnico, LRR-Technish Universiät München and MCS-Emory University, 1997.

[89] B. M. Maggs, L. R. Matheson, R. E. Tarjan. *Models of Parallel Computation: A Survey and Synthesis*. In *HICSS'95*. IEEE, Inc., Jan. 1995.

[90] E. Maillet. *Le traçage logiciel d'applications parallèles: conception et ajustement de qualité*. Tese de Doutoramento, Institut National Polytechnique de Grenoble, 1996.

[91] D. C. Marinescu, J. James E. Lumpp, T. L. Casavant, H. J. Siegel. *Models for Monitoring and Debugging Tools for Parallel and Distributed Software*. *Journal of Parallel and Distributed Computing*, (9):171–184, 1990.

[92] R. Marques, J. C. Cunha. *PVM-Prolog User's Guide*. Departamento de Informática, FCT-Universidade Nova de Lisboa, 1995.

[93] R. Marques, J. C. Cunha. *Using PVM with Logic Programming Interface*. In *Proc. of 2nd EuroPVM User's Meeting*, Lyon, France, 1995.

[94] F. Mattern. *Virtual Time and Global States of Distributed Systems*. In M. Cosnard et al., editores, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pág. 215–226, Amsterdam, 1989. Elsevier Science Publishers.

[95] P. J. Mercurio, T. T. Elvins, S. J. Young, P. S. Cohen, K. R. Fall, M. H. Ellisman. *The Distributed Laboratory, An interactive visualization environment for electron microscopy and 3D imaging*. *Communications of the ACM*, 35(6):55–63, Jun. 1992. SIG Graph'92 Showcase.

[96] B. P. Miller, J. K. Hollingsworth, M. D. Callaghan. *The Paradyn Parallel Performance Measurement Tools*. *IEEE Computer*, pág. 37–46, nov. 1995. Special issue on performance evaluation tools for parallel and distributed computer systems.

[97] B. P. Miller. *What to Draw? When to Draw? An Essay on Parallel Program Visualization*. *Journal of Parallel and Distributed Computing*, (18):265–269, 1993.

[98] B. Mohr. *Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?* In J. Dongarra, B. Tourabcheau, editores, *Workshop Environments and Tools for Scientific Parallel Computing*, vol. 6, pág. 103–124. Advances in Parallel Computing, Elsevier, 93.

[99] B. Moscão, D. Pereira, J. Vieira. *Distributed Applications Monitoring System*. Relatório técnico, Dep. de Informática, FCT/UNL, 1998. Projecto de fim de curso.

[100] A. Mostefaoui, O. Theel. *Reduction of Timestamp Sizes for Causal Event Ordering*. Relatório técnico 1062, IRISA, Rennes-France, Nov. 96. Projet ADP.

[101] MPI Forum. *MPI: A Message-Passing Interface Standard*. University of Tennessee, 1995.

[102] MPI Forum. *MPI-2: Extensions to the Message-Passing Interface*. University of Tennessee, 1997.

[103] J. Muthukumarasamy, J. T. Stasko. *Visualizing Program Executions on Large Data Sets Using Semantic Zooming*. Relatório técnico GIT-GVU-95-02, GVUC/CC-Georgia Institute of Technology, 95.

[104] R. H. B. Netzer, B. P. Miller. *Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs*. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[105] M. Oberhuber, R. Wismuller. *DETOP - An Interactive Debugger for PowerPC Based Multicomputers*. *Parallel Programming and Applications*, pág. 170–183, 1995.

[106] Object Management Group. *The Common Object Request Broker: Architecture and Specification (v2.4)*, 2000.

[107] The Open Group. *The Single UNIX Specification, Version 2*, 1997.

[108] *PADIPRO – Parallel Distributed Prolog and Applications*, 1996. Relatório final, Projecto Digital Equipment Corporation European External Research Programme.

[109] *Parallel Processing Tools: Integration and Dissemination*. INCO 977100, "Keep In Touch"(KIT) do programa Copernicus, 1997–1999.

[110] D. M. Pase. *Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide*. Relatório técnico, IBM, 1998.

[111] R. Prodan, J. M. Kewley. *A Framework for an Interoperable Tool Environment*. In *Proc. Euro-Par 2000*, vol. 1900 de *LNCS*, pág. 65–69. Springer, 2000.

[112] *PROLOPPE – Programação em Lógica com Extensões*, 1997. Relatório final, Projecto PRAXIS XXI.

[113] *PTLib – Parallel Tools Library, NHSE*. `http://rib.cs.utk.edu/catalog.pl?rh=223`.

[114] *PTools – The Parallel Tools Consortium*. `http://www.ptools.org/`.

[115] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. Schwartz, L. F. Tavera. *Scalable Performance Analysis: The Pablo Performance Analysis Environment*. In *Proc. of the Scalable Parallel Libraries Conference*, pág. 104–113. IEEE Computer Society, 1993.

[116] *SETNA-ParComp — Scalable Environments and Parallel Computing*, 2000. Relatório final, Projecto PRAXIS XXI.

[117] D. B. Skillicorn, D. Talia. *Models and Languages for Parallel Computation*. Relatório técnico, Queen's University, Kigston, Canada Universitá della Calabria, Rende, Italy, 1996.

[118] R. Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2*. In *RFC 1831*. Network Working Group, 1995.

[119] J. T. Stasko. *The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report*. Relatório técnico GIT-GVU-95-03, GVUC/CC - Georgia Institute of Technology, 95.

[120] SUN-Microsystems. *Java*. `http://java.sun.com/`.

[121] A. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[122] *TotalView*. Dolphin ToolWorks, Dolphin Interconnect Solutions, Inc., Framingham, Massachusetts, USA.

[123] J. Trinitis, V. Sunderam, T. Ludwig, R. Wismüller. *Interoperability Support in Distributed On-line Monitoring Systems*. In *Proc. of the International Conference on High-Performance Computing and Networking (HPCN'2000), Springer-Verlag*, vol. 1823, Amsterdam, The Netherlands, 2000.

[124] R. Vaz, N. Neves. *Steering de algoritmos genéticos paralelos*. Relatório técnico, Dep. de Informática, FCT/UNL, 1998. Projecto de fim de curso.

[125] X.-F. Vigouroux. *Analyse Distribuée de Traces d'Exécution de Programmes Parallèles*. Tese de Doutoramento, L'École Normale Supérieure de Lyon, Jan. 96.

[126] W3C. *Extensible Markup Language (XML) 1.0*. Relatório técnico, World Wide Web Consortium, 2000.

[127] R. Wismüller, T. Ludwig. *Interoperable Run-Time Tools for Distributed Systems – A Case Study*. In H. R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. IV, pág. 1763–1769, Las Vegas, USA, Jun. 1999. CSREA Press.

[128] R. Wismüller, M. Oberhuber, J. Krammer. *Interactive Debugging and Performance Analysis of Massively Parallel Applications*. Relatório técnico 22, 1996.

[129] P. H. Worley. *A New PICL Trace File Format*. Relatório técnico ORNL/TM-12125, Oak Ridge National Laboratory, Tennessee, 1992.

[130] B. J. N. Wylie, A. Endo. *Annai/PMA Multi-level Hierarchical Parallel Program Performance Enginnering*. Relatório técnico TR-96-03, SCSC, Manno, Switzerland, 1996.